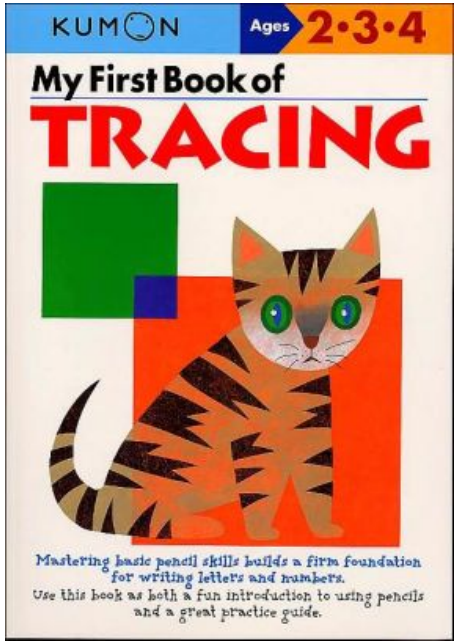
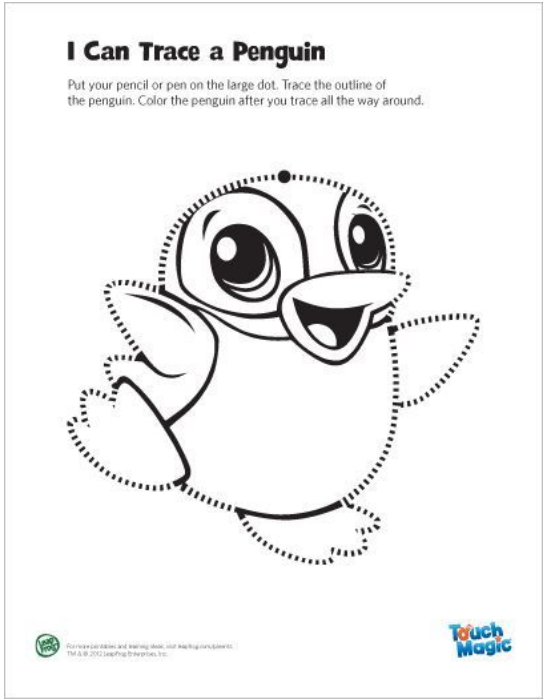


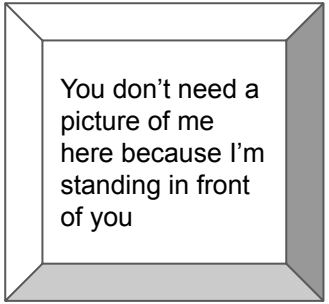
Linux Tracing

Anthony Critelli



All views and information expressed in this presentation are my own, and do not represent the views or opinions of any employer, present, past, or future.

About me



- Linux systems engineer
- Dislike colorful slides
- Started career as a network engineer
  - Not much interesting going on in networking (other than marketing buzzwords)
  - Lots interesting going on in Linux
- Interests
  - Automation
  - Occasionally blog (both on my personal site and for RedHat Enable SysAdmin)
- “Hey, you look familiar”
  - Former IST (the school formerly known as NSSA now known as iSchool) sysadmin
  - Former CCDC coach (0-1 record, they fired me. JK, I stopped working at RIT)

What do I mean by tracing?

- Observing the execution of an arbitrary program, including the kernel itself, as its running
- In particular, I'd like to:
  - Attach to a live program
  - Have low overhead
- There are *a lot* of tracers for Linux
  - **http://www.brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html**
- We'll focus on two tracing tools today, just because:
  - Ftrace
  - eBPF, specifically the BCC project

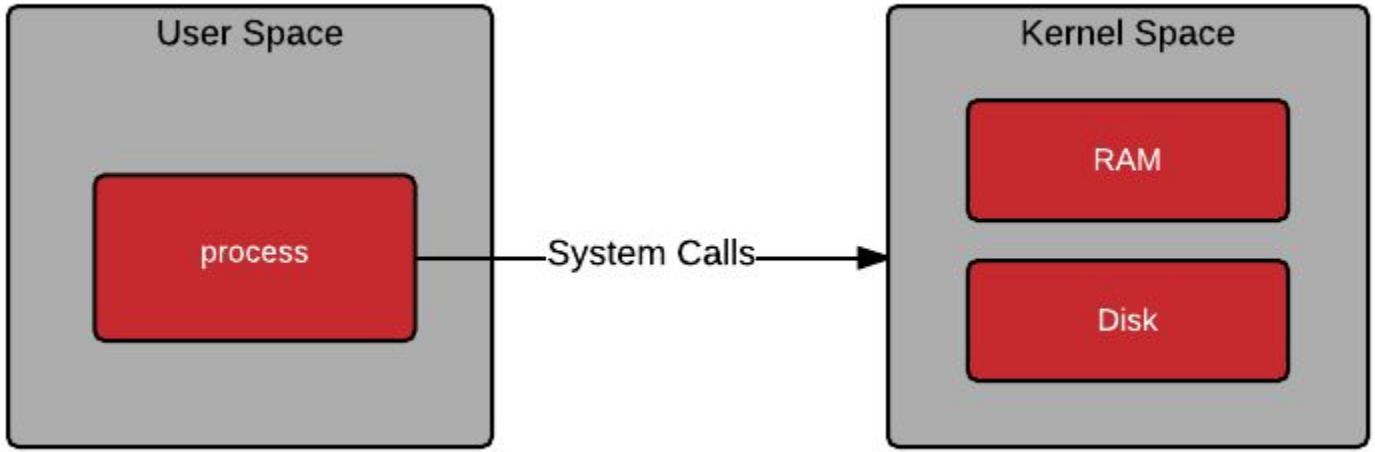
Why might you trace?

- Troubleshooting - many userland programs have trace probes (or can be compiled with them).
  - Example: MySQL
- Performance analysis - understand what your code is doing and where it might be bottlenecking, among other things
- For the learning opportunity - start tracing a user or kernel function, and you'll find yourself down the rabbit hole of trying to better understand the codebase
- Demystify the black box. Computers really aren't magic.
  - Except they kind of are. I mean, this is all a bunch of sand doing math real quick so that you can shitpost on Workplace

What do I need before I start?

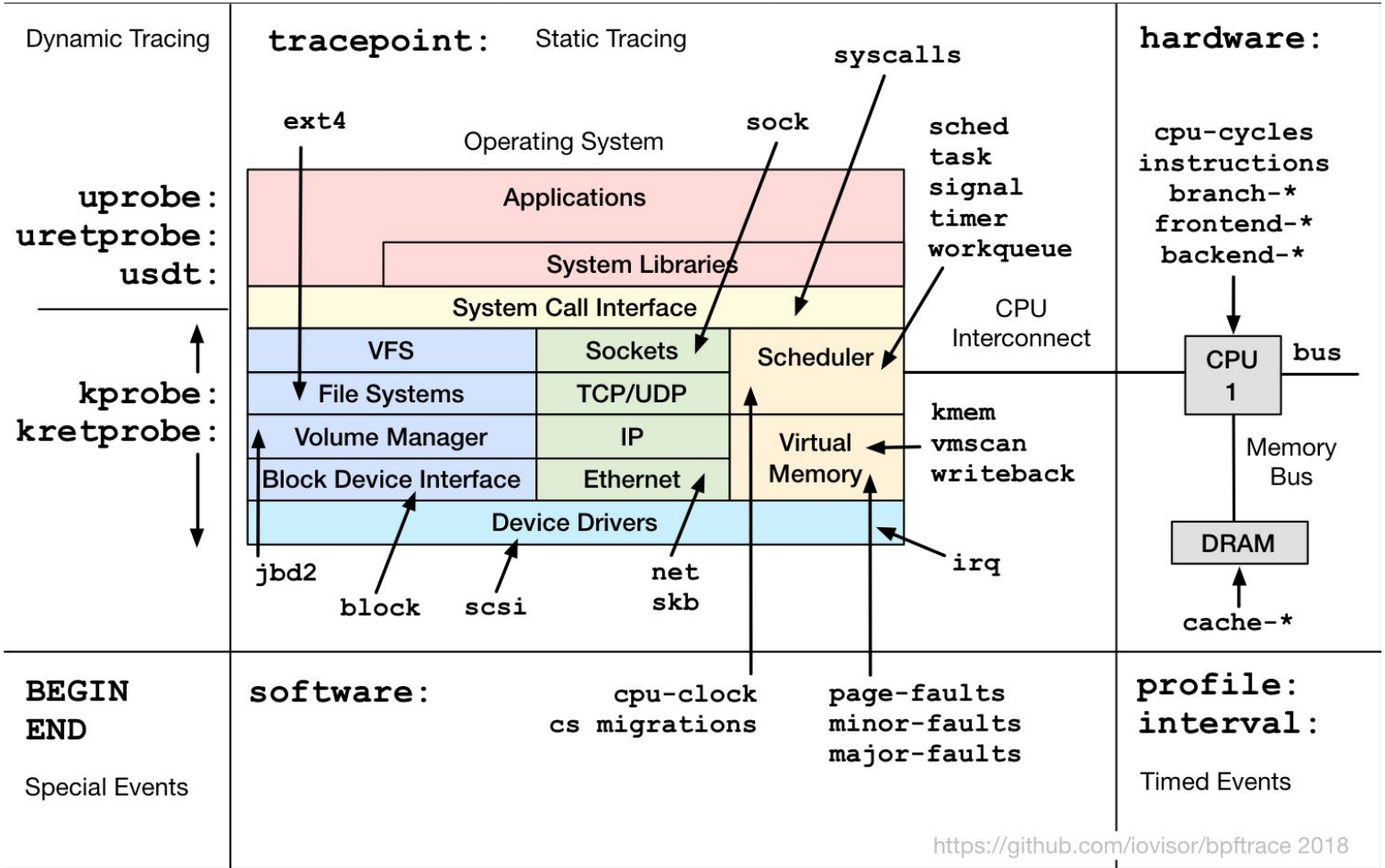
- A basic knowledge of C, especially a basic knowledge of pointers
  - You can start playing around without this, but it helps *a lot*
  - Trust me: just take the 15 minutes needed to refresh your knowledge of pointers
- A basic understanding of how Linux works
  - User vs. kernel space, etc.
- Willingness to be very out of your element, unless you do a lot of programming in C, or kernel development
  - There are a lot of rabbit holes to follow

User Space vs. Kernel Space



https://www.redhat.com/en/blog/architecting-containers-part-1-why-understanding-user-space-vs-kernel-space-matters

bpftrace Probe Types



Tools

- Many tools, but we'll look at Ftrace and eBPF (specifically, the BCC tools)
- Why?
  - It's what I'm personally "comfortable" with, and I'm the one talking at you
- Ftrace has a low barrier to entry, you can immediately start looking at what your kernel is doing
- eBPF is really gaining popularity
  - Highly espoused by our lord and savior Brendan Gregg
- I've found that using a combination of Ftrace (for initially looking at things) and eBPF/BCC have allowed me to start tracing interesting things

Terminology: Static vs. Dynamic? User vs. Kernel?

- Static tracing - the program’s source code must be instrumented to expose a probe
  - Requires modifying the code
  - Many programs do expose some static trace probes
- Dynamic tracing - source code modification isn’t necessary
  - Often just requires installing debugging symbols
- User tracing - tracing of code that executed in userspace
- Kernel tracing - tracing of code (the kernel and its functions) that run in kernel space

Ftrace

https://www.kernel.org/doc/Documentation/trace/ftrace.txt

- Kernel **f**unction **tracer** (actually a collection/framework of tracing tools)
- Has many useful built-in tracers: hardware latency, function graphing, interrupt requests, block I/O, etc.
- You can interact with it like a file system at /sys/kernel/tracing or /sys/kernel/debug/tracing
    - There’s also a front-end called trace-cmd

eBPF

- Grew out of BPF (Berkeley packet filters)
  - Wait what? Packets?
  - Yeah: turns out it's useful to run some user code inside the kernel
- eBPF is an in-kernel "virtual machine"
  - That is: the eBPF code compiles down to its own bytecode that the kernel executes
  - No, it's not a Windows XP VM running in your kernel
- It's fast: Netflix runs some of their instrumentation in production
- Writing eBPF bytecode is miserable (like, you know: writing any kind of bytecode)
  - I've never done this
  - Tools have been developed to provide front-ends

The eBPF Compiler collection (BCC)

https://github.com/iovisor/bcc

- A set of tools to make writing and interacting with eBPF easier
  - Write the kernel instrumentation in C
  - Write frontends in Python
- Why is this neat?
  - You can do kernel eBPF things in C, as you would expect
  - You can then send this data to userspace Python scripts
  - It's *much* easier to work with data in Python
    - Still be considerate of how often things are getting sent to userspace (expensive)

BCC Continued

- BCC has some nice things to make your life easier
- Functions for writing to shared tracing pipe (same one used by ftrace)
- Macros and functions for things like:
  - Maintaining hashes that can later be accessed by userspace
  - Building histograms, also accessible in userspace later
- The repository also has a wealth of existing, pre-written tools
  - These can probably do most things you'd be interested in
  - Great examples of how to write your own programs

BPF Trace

https://github.com/iovisor/bpftrace/

- A higher level tracing language on top of eBPF and BCC
- Great for writing one-liners
  - Lots of examples on their GitHub
- Won’t be discussing it further here

(some) Things that you can do with eBPF and these tools

- Dynamically trace kernel functions (entry and return) and get their arguments
  - Remember, dynamically just means that the kernel doesn't need any special trace points to support this
- Do the same thing for userspace applications
  - Typically, you'd install the debugging symbols/recompile a package with debugging enabled for the binary that you're interested in playing with
- Do the same thing for syscalls
- Perform these same operations with static trace points
  - Many applications expose this kind of instrumentation

Front end	Difficulty	Pros	Cons	References
BPF bytecode	Brutal	Precise control	Insanely difficult	Kernel source: struct bpf_insn prog in samples/bpf/sock_example.c
C	Hard	Build stand-alone binaries	Difficult	Kernel source: samples/bpf/tracex1_kern.c and samples/bpf/tracex1_user.c
perf	Hard	Use perf's capabilities: custom events, stack walking	Difficult, not yet well documented	Section below: 7. perf.
bcc	Moderate	Custom output, Python libraries, large community, production use (e.g., Facebook, Netflix)	Verbose	Section below: 5. bcc.
bpftrace	Easy	Powerful one-liners, many capabilities, growing community, production use (e.g., Netflix, Facebook)	Limited control of code and output	Section below: 6. bpftrace.
ply	Easy	Powerful one-liners, small binary, for embedded	Limited control of code and output	github: github.com/iovisor/ply.

http://www.brendangregg.com/ebpf.html

So, where do I start?

- Learning where to start can be daunting, especially if you just want to learn and don’t have a defined use case
- What I found useful:
  - Pick some arbitrary thing, like doing a DNS lookup with nslookup
  - Observe it using ftrace.
  - See what kernel functions are called
  - Pick some of the more obvious ones (e.g. network, I/O, filesystem, etc.)
  - See if you can write a BCC program to collect some data (how many times the function is called, its arguments, etc.)
- Hopefully, this presentation will give you some ideas in practice
- Everyone learns differently: honestly, it’s probably better to read the example code first, then dive in

Live Demo

Tips and Tricks for Learning

- All of these tools have lots of examples
  - The eBPF ecosystem, in particular, has tons of pre-written tools and examples
- I found the BCC developer tutorial to be helpful (check the BCC docs folder)
- While the eBPF tools provide a lot of features, none of this is really “batteries included”
  - Expect to try and fail a lot
  - Be aware of differences between eBPF versions (e.g. you might be reading docs for a newer version)
- Just get started: find something that sounds neat, and learn to explore it

A few resources

- Review the tools and examples in the previously repositories
- Brendan Gregg's blog http://www.brendangregg.com
- Bootlin's Elixir tool for browsing kernel source: https://elixir.bootlin.com/linux/latest/source
- A packet's journey through the kernel: https://blog.packagecloud.io/eng/2017/02/06/monitoring-tuning-linux-networking-stack-sending-data/
- Julia Evan's blog: https://jvns.ca
- A bunch of eBPF resources: https://qmonnet.github.io/whirl-offload/2016/09/01/dive-into-bpf/

Thanks! Questions?

critellia@gmail.com
www.acritelli.com