ftrace: trace your kernel functions!

Hello! Today we're going to talk about a debugging tool we haven't talked about much before on this blog: ftrace. What could be more exciting than a new debugging tool?!

Better yet, ftrace isn't new! It's been around since Linux kernel 2.6, or about 2008. here's the earliest documentation I found with some quick Gooogling. So you might be able to use it even if you're debugging an older system!

I've known that ftrace exists for about 2.5 years now, but hadn't gotten around to really learning it yet. I'm supposed to run a workshop tomorrow where I talk about ftrace, so today is the day we talk about it!

what's ftrace?

ftrace is a Linux kernel feature that lets you trace Linux kernel function calls. Why would you want to do that? Well, suppose you're debugging a weird problem, and you've gotten to the point where you're staring at the source code for your kernel version and wondering what **exactly** is going on.

I don't read the kernel source code very often when debugging, but occasionally I do! For example this week at work we had a program that was frozen and stuck spinning inside the kernel. Looking at what functions were being called helped us understand better what was happening in the kernel and what systems were involved (in that case, it was the virtual memory system)!

I think ftrace is a bit of a niche tool (it's definitely less broadly useful and harder to use than strace) but that it's worth knowing about. So let's learn about it!

first steps with ftrace

Unlike strace and perf, ftrace isn't a **program** exactly – you don't just run `ftrace my_cool_function`. That would be too easy!

If you read Debugging the kernel using Ftrace it starts out by telling you to `cd /sys/kernel/debug/tracing` and then do various filesystem manipulations.

For me this is way too annoying – a simple example of using ftrace this way is something like

```
cd /sys/kernel/debug/tracing
echo function > current_tracer
echo do_page_fault > set_ftrace_filter
cat trace
```

This filesystem interface to the tracing system ("put values in these magic files and things will happen") seems theoretically possible to use but really not my preference.

Luckily, team ftrace also thought this interface wasn't that user friendly and so there is an easier-to-use interface called **trace-cmd**!!! trace-cmd is a normal program with command line arguments. We'll use that! I found an intro to trace-cmd on LWN at trace-cmd: A front-end for Ftrace.

getting started with trace-cmd: let's trace just one function

First, I needed to install `trace-cmd` with `sudo apt-get install trace-cmd`. Easy enough.

For this first ftrace demo, I decided I wanted to know when my kernel was handling a page fault. When Linux allocates memory, it often does it lazily ("you weren't *really* planning to use that memory, right?"). This means that when an application tries to actually write to memory that it allocated, there's a page fault and the kernel needs to give the application physical memory to use.

Let's start `trace-cmd` and make it trace the `do_page_fault` function!

```
$ sudo trace-cmd record -p function -l do_page_fault
  plugin 'function'
Hit Ctrl^C to stop recording
```

I ran it for a few seconds and then hit `Ctrl+C`. Awesome! It created a 2.5MB file called `trace.dat`. Let's see what's that file!

```
$ sudo trace-cmd report
          chrome-15144 [000] 11446.466121: function:             do_page_fault
          chrome-15144 [000] 11446.467910: function:             do_page_fault
          chrome-15144 [000] 11446.469174: function:             do_page_fault
          chrome-15144 [000] 11446.474225: function:             do_page_fault
          chrome-15144 [000] 11446.474386: function:             do_page_fault
          chrome-15144 [000] 11446.478768: function:             do_page_fault
 CompositorTileW-15154 [001] 11446.480172: function:             do_page_fault
          chrome-1830  [003] 11446.486696: function:             do_page_fault
 CompositorTileW-15154 [001] 11446.488983: function:             do_page_fault
 CompositorTileW-15154 [001] 11446.489034: function:             do_page_fault
 CompositorTileW-15154 [001] 11446.489045: function:             do_page_fault
```

This is neat – it shows me the process name (chrome), process ID (15144), CPU (000), and function that got traced.

By looking at the whole report, (`sudo trace-cmd report | grep chrome`) I can see that we traced for about 1.5 seconds and in that time Chrome had about 500 page faults. Cool! We have done our first ftrace!

next ftrace trick: let's trace a process!

Okay, but just seeing one function is kind of boring! Let's say I want to know everything that's happening for one program. I use a static site generator called Hugo. What's the kernel doing for Hugo?

Hugo's PID on my computer right now is 25314, so I recorded all the kernel functions with:

```
sudo trace-cmd record --help # I read the help!
sudo trace-cmd record -p function -P 25314 # record for PID 25314
```

`sudo trace-cmd report` printed out 18,000 lines of output. If you're interested, you can see all 18,000 lines here.

18,000 lines is a lot so here are some interesting excerpts.

This looks like what happens when the `clock_gettime` system call runs. Neat!

```
 compat_SyS_clock_gettime
    SyS_clock_gettime
       clockid_to_kclock
       posix_clock_realtime_get
          getnstimeofday64
             __getnstimeofday64
                arch_counter_read
    __compat_put_timespec
```

This is something related to process scheduling:

```
 cpufreq_sched_irq_work
    wake_up_process
       try_to_wake_up
          _raw_spin_lock_irqsave
             do_raw_spin_lock
          _raw_spin_lock
             do_raw_spin_lock
          walt_ktime_clock
             ktime_get
```

```
                arch_counter_read
          walt_update_task_ravg
             exiting_task
```

Being able to see all these function calls is pretty cool, even if I don't quite understand them.

"function graph" tracing

There's another tracing mode called `function_graph`. This is the same as the function tracer except that it instruments both entering *and* exiting a function. Here's the output of that tracer

```
sudo trace-cmd record -p function_graph -P 25314
```

Again, here's a snipped (this time from the futex code)

```
              |      futex_wake() {
              |        get_futex_key() {
              |          get_user_pages_fast() {
  1.458 us    |            __get_user_pages_fast();
  4.375 us    |          }
              |          __might_sleep() {
  0.292 us    |            ___might_sleep();
  2.333 us    |          }
  0.584 us    |          get_futex_key_refs();
              |          unlock_page() {
  0.291 us    |            page_waitqueue();
  0.583 us    |            __wake_up_bit();
  5.250 us    |          }
  0.583 us    |          put_page();
+ 24.208 us   |        }
```

We see in this example that `get_futex_key` gets called right after `futex_wake`. Is that what really happens in the source code? We can check!! Here's the definition of futex_wake in Linux 4.4 (my kernel version).

I'll save you a click: it looks like this:

```
static int
futex_wake(u32 __user *uaddr, unsigned int flags, int nr_wake, u32 bitset)
{
        struct futex_hash_bucket *hb;
        struct futex_q *this, *next;
        union futex_key key = FUTEX_KEY_INIT;
        int ret;
        WAKE_Q(wake_q);

        if (!bitset)
                return -EINVAL;

        ret = get_futex_key(uaddr, flags & FLAGS_SHARED, &key, VERIFY_READ);
```

So the first function called in `futex_wake` really is `get_futex_key`! Neat! Reading the function trace was definitely an easier way to find that out than by reading the kernel code, and it's nice to see how long all of the functions took.

How to know what functions you can trace

If you run `sudo trace-cmd list -f` you'll get a list of all the functions you can trace. That's pretty simple but it's important.

one last thing: events!

So, now we know how to trace functions in the kernel! That's really cool!

There's one more class of thing we can trace though! Some events don't correspond super well to function calls. For example, you might want to know when a program is scheduled on or off the CPU! You might be able to figure that out by peering at function calls, but I sure can't.

So the kernel also gives you a few events so you can see when a few important things happen. You can see a list of all these events with `sudo cat /sys/kernel/debug/tracing/available_events`

I looked at all the sched_switch events. I'm not exactly sure what sched_switch is but it's something to do with scheduling I guess.

```
sudo cat /sys/kernel/debug/tracing/available_events
sudo trace-cmd record -e sched:sched_switch
sudo trace-cmd report
```

The output looks like this:

```
 16169.624862:   Chrome_ChildIOT:24817 [112] S ==> chrome:15144 [120]
 16169.624992:   chrome:15144 [120] S ==> swapper/3:0 [120]
 16169.625202:   swapper/3:0 [120] R ==> Chrome_ChildIOT:24817 [112]
 16169.625251:   Chrome_ChildIOT:24817 [112] R ==> chrome:1561 [112]
 16169.625437:   chrome:1561 [112] S ==> chrome:15144 [120]
```

so you can see it switching from PID 24817 -> 15144 -> kernel -> 24817 -> 1561 -> 15114. (all of these events are on the same CPU)

how does ftrace work?

ftrace is a dynamic tracing system. This means that when I start ftracing a kernel function, the **function's code gets changed**. So – let's suppose that I'm tracing that `do_page_fault` function from before. The kernel will insert some extra instructions in the assembly for that function to notify the tracing system every time that function gets called. The reason it can add extra instructions is that Linux compiles in a few extra NOP instructions into every function, so there's space to add tracing code when needed.

This is awesome because it means that when I'm not using ftrace to trace my kernel, it doesn't affect performance at all. When I do start tracing, the more functions I trace, the more overhead it'll have.

(probably some of this is wrong, but this is how I think ftrace works anyway)

use ftrace more easily: brendan gregg's tools & kernelshark

As we've seen in this post, you need to think quite a lot about what individual kernel functions / events do to use ftrace directly. This is cool, but it's also a lot of work!

Brendan Gregg (our linux debugging tools hero) has repository of tools that use ftrace to give you information about various things like IO latency. They're all in his perf-tools repository on GitHub.

The tradeoff here is that they're easier to use, but you're limited to things that Brendan Gregg thought of & decided to make a tool for. Which is a lot of things! :)

Another tool for visualizing the output of ftrace better is kernelshark. I haven't played with it much yet but it looks useful. You can install it with `sudo apt-get install kernelshark`.

a new superpower

I'm really happy I took the time to learn a little more about ftrace today! Like any kernel tool, it'll work differently between different kernel versions, but I hope that you find it useful one day.

an index of ftrace articles

Finally, here's a list of a bunch of ftrace articles I found. Many of them are on LWN (Linux Weekly News), which is a pretty great source of writing on Linux. (you can buy a subscription!)

- Debugging the kernel using Ftrace - part 1 (Dec 2009, Steven Rostedt)
- Debugging the kernel using Ftrace - part 2 (Dec 2009, Steven Rostedt)
- Secrets of the Linux function tracer (Jan 2010, Steven Rostedt)
- trace-cmd: A front-end for Ftrace (Oct 2010, Steven Rostedt)
- Using KernelShark to analyze the real-time scheduler (2011, Steven Rostedt)
- Ftrace: The hidden light switch (2014, Brendan Gregg)
- the kernel documentation: (which is quite useful) Documentation/ftrace.txt
- documentation on events you can trace Documentation/events.txt
- some docs on ftrace design for linux kernel devs (not as useful, but interesting) Documentation/ftrace-design.txt