Fantastic Tracepoints and Where to Find Them

BKK-TR04

Daniel Thompson, Linaro Support and Solutions Engineering

April 2019

ftrace

Demonstrates: *ftrace*, *debugfs interface*

Let's start with a super quick example of how to control tracing directly from debugfs.

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

The `trace` file is non-destructive but there is also `trace_pipe` which is good for watching live systems and removes each trace entry that has been passed on to the cat process:

```
echo 0 > tracing_on
cat trace
cat trace_pipe
cat trace
```

reset

Demonstrates: *How not to get lost!*

This is a live demo and I'm worried I might get lost so I'll just setup a quick macro that, more or less, puts us back as we were after reboot

```
trace-reset () {
    cd /sys/kernel/debug/tracing
    echo 0 > tracing_on
    echo nop > current_tracer
    echo 0 > events/enable
    echo "" > kprobe_events
    echo 1 > tracing_on
    echo > trace
}
trace-reset
```

Tracepoints

Demonstrates: *Fantastic Tracepoints and Where to Find Them*

That's pretty much all we are going to say about ftrace today. To be clear ftrace is *seriously* awesome and I could easily fill the entire of today's session talking about filtering, triggers and advanced tracers. That would also be a great talk but I sent in an Abstract that promised to talk about tracepoints so that is where we are going to go from here.

```
less README
find events -type d
```

That's it... we can all go home!

regmap

Demonstrates: *tracepoints, test glob filtering, numeric filtering*

Regmap is an interested sub-system. It's genesis was as a means to abstract the bus used to access registers for devices that support multiple register access protocols. For example a temperature sensor that can be interfaced with either I2C or SPI. Using regmap to abstract the accessors allows the same driver to support both modes with minimal effort.

However regmap has been extended with multiple protocols, including memory mapped I/O, where it is great for coping with peripherals who reorganised the register layout in version 2 but without changing the feature set much.

Anyhow, regmap gives the driver a lot of things for free. For example it reveals the register state via debugfs... and each accessor contains a tracepoint.

```
ls /sys/kernel/debug/regmap
```

regmap - Lenovo Yoga C630 (arm64)

```
# Uses regmap to access PMIC registers
echo 1 > events/regmap/enable
cat trace_pipe

cat events/regmap/regmap_reg_write/format
echo 'name ~ "*thermal*"' > events/regmap/filter
cat trace_pipe
cat events/regmap/filter
cat events/regmap/regmap_reg_write/filter

cat events/regmap/regmap_reg_write/format
echo 'name ~ "*thermal*" && reg == 0xc0' > events/regmap/filter
cat trace_pipe
```

regmap - Lenovo Yoga 910 (x86_64)

Note: *This will not be presented... unless there is trouble with the WiFi...*

```
# Uses regmap to abstract codec registers
echo 1 > events/regmap/enable
cat trace_pipe

cat events/regmap/regmap_reg_write/format
echo 'name ~ "hdaudio*"' > events/regmap/filter
echo > trace
cat trace_pipe
cat events/regmap/filter
cat events/regmap/regmap_reg_write/filter

echo 'name ~ "notaudio*"' > events/regmap/filter
echo > trace
cat trace-pipe
```

i2c

Demonstrates: *tracepoints*, *numeric filtering*, *triggers*, *histograms*

Moving on from regmap I thought we'd spend a moment looking at the i2c tracepoints. I picked these because I hope you will all find it fairly easy to understand how the i2c bus works. I can also mention that on the (64-bit Arm) laptop running this demo then the keyboard, trackpad and touchscreen are all connecting using the hid-over-i2c protocol which makes it easy for me to provoke i2c activity on this system.

```
echo 1 > events/i2c/enable
cat trace_pipe

# Examine output and remember which adapter is keyboard, trackpad, etc
kb=11
tp=3
```

i2c - Continued

```
cat events/i2c/i2c_reply/format
echo 'adapter_nr == $tp' > events/i2c/filter
cat trace_pipe

echo 'adapter_nr == $kb' > events/i2c/filter
echo 'stacktrace' > events/i2c/i2c_read/trigger
cat trace_pipe
echo '!stacktrace' > events/i2c/i2c_read/trigger

cat events/i2c/i2c_read/format
echo 'hist:keys=stacktace,adapter_nr'
watch cat events/i2c/i2c_read/hist
echo '!hist'
```

Dynamic probes

Demonstrates: *dynamic tracepoints*, *numeric filtering*, *errors in filtering*

So far we have relied entirely upon static tracepoints built into the kernel. There are lots of them and I encourage you to look though the `events` directory on your laptop or target devices and look and see what interests you.

However the real secret of where to find Fantastic Tracepoints is that you can dynamically add tracepoints almost anywhere in the kernel meaning the ultimate tool for findinng tracepoints is the source code.

In this example we're going to trace VFS functions. Again this is because the relationship between what I type at the keyboard and which VFS functions will be used should be fairly clear to you.

Dynamic probes - Continued

```
echo 'p:vfs_read __vfs_read' > kprobe_events
echo 1 > events/kprobes/enable
cat trace_pipe

echo 'r:vfs_read_retval __vfs_read $retval' >> kprobe_events
echo 1 > events/kprobes/enable
cat trace_pipe

echo 0 > events/kprobes/enable
echo 'p:vfs_read __vfs_read iname=+56(+24(%x0)):string count=%x2:x64' \
        > kprobe_events
cat events/kprobes/vfs_read/format
cat trace_pipe
```

Bonus Puzzle

This will not be presented...

Steven Rostedt shared this at ELCE18... what does it do?

[requires a "sane" 64-bit kernel without structure randomization for the offsets to work... the offsets will eventually drift; I last tested it on v4.20)]

```
echo 'p:crazy __vfs_read name=+0(+0(+40(+40(+32($arg1))))):string' \
        > kprobe_events

# If you don't have argX support in your kernel but you *do* have an
# arm64 kernel then try this instead
echo 'p:crazy __vfs_read name=+0(+0(+40(+40(+32(%x0))))):string' \
        > kprobe_events
```

Dynamic probes with perf probe

Just stop and reflect for a second that everything I have shown you so far has been based on basic tools that can be found in almost all Linux based systems, from fully fledged GNU/Linux distros through to embedded systems based on busybox/musl or Android. Not only that but if you need a memory jogger to remind you how the trace system works the mini-HOWTO is built into the kernel: `/sys/kernel/debug/tracing/README` .

Anyhow... we've finished doing it by hand. Let's see what happens if we bring in a few tools to help us.

In this case we're going to use perf's symbolic debug facilities to create tracepoints without having to resort to machine level debug techniques.

Dynamic probes with perf probe - Continued

```
perf probe \
    '__vfs_read iname=file->f_path.dentry->d_iname:string count=count'
echo 1 > events/probe/__vfs_read/enable
cat trace_pipe
echo 0 > events/enable
perf probe -d __vfs_read
```

There is even cross-tooling support that allows `perf probe` to do the symbolic work on a development workstation and generate probe strings ready to copy to an under resourced target machine.

```
perf probe -D \
    '__vfs_read iname=file->f_path.dentry->d_iname:string count=count'
perf probe --vmlinux ~drt/Development/Kernel/linux/vmlinux -D \
    '__vfs_read iname=file->f_path.dentry->d_iname:string count=count'
```

Tracepoints and ply

To close out the session I'd like to when two powerful kernel tools meet each other: in this example let's see what happens when dynamic tracepoints and eBPF, the kernel's built-in virtual machine are introduced to each other.

We'll be using the lightweight `ply` tool for this. There are larger and more powerful eBPF toolchains out there... but `ply` has no dependancies and should be easy to get working on whatever distribution you find yourself. `ply` is an `awk`-like mini-language that allows us to attach code to one or more tracepoints.

Tracepoints and ply - part 1

Firstly lets try a simple one-liner

```
ply -t 5 -c 'kprobe:kmem_cache_alloc_node { @[stack()].count() }'
```

If you are paying attention you'll be getting a sense of deja-vu here... this is very similar to the histogram triggers we attached to the i2c_read tracepoint.

Tracepoints and ply - part 2

So... let's look at something we can't do with tracepoints alone.

```
cd $HOME
cat > track.ply <<EOF
kprobe:kmem_cache_alloc_node {
# Can't read stack from a retprobe :-(
    @[0] = stack();
}
kretprobe:kmem_cache_alloc_node {
    @[retval()] = @[0];
    @[0] = nil;
}
kprobe:kmem_cache_free {
    @[arg(1)] = nil;
}
EOF
ply track.ply
```

Thanks

THANK YOU!

Any questions about this presentation? support@linaro.org

Any questions about any other Linaro activity? (if you work for a member company) support@linaro.org

With special thanks to: Steven Rostedt, Masami Hiramatsu and Leo Yan