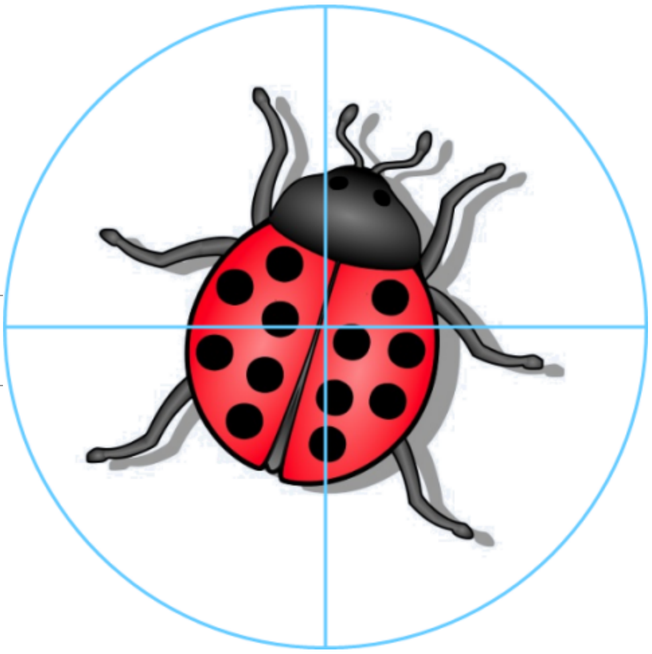
How to debug a Linux driver using FTrace

Call: +44 (0)23 8098 8890
E-mail: sales@itdev.co.uk (mailto:sales@itdev.co.uk)

Posted 3rd August 2020, By James H (/company/about/team/james-h)

Introduction



In this blog I will be discussing debugging Linux drivers using `ftrace` as an alternative to the most basic, yet surprisingly prevalent, method of `printk` debugging.

As with many things in Linux, the documentation (https://elixir.bootlin.com/linux/latest/source/Documentation/trace/ftrace.rst) is good, so do go and have a read. It introduces `ftrace` as follows:

> Ftrace is an internal tracer designed to help out developers and designers of systems to find what is going on inside the kernel.

We will focus on the function tracer, which will allow us to see the paths that our system calls take through the kernel.

To demonstrate this part of the `ftrace` functionality, I have written a very basic character device driver that provides a simple problem to solve. A character device has been used as this doesn't require us to modify anything in our kernel or the device tree to get it running. We also don't have to get into any of the Linux driver framework. It will just load and be usable and thus you can use it on your desktop Linux machine without having to use any other hardware.

The idea behind the driver is to imagine some device that provides data of some kind that must be read separately from the character driver's generic read system call, through a `sysfs` binary file. The device always has some information to pass on so we can assume that reading this file never blocks and acts as if it is "infinite" in length. The data read could be anything, but the point is the imagined author of this driver has made a slight error and on testing sees that a read of one data block from this file works, but all subsequent reads do not. I'll refer to the author as "she" to avoid repeatedly writing 'he' or 'she'.

For some quick debug she uses a `printk()` to examine calls to the file's read function. The first read outputs a message to the system log, but the second read does not. She is perplexed... why is the read function only being called once?! A quick inspection of the driver code hasn't revealed anything obvious to her.

Without much knowledge of the internals of the kernel and its `sysfs` framework, a driver author might feel a little at a loss. But for this author it is not so, because she knows that `ftrace` can be used to find out what's happening.

Setting up the Example

The first thing to do is clone our repository for this blog:

```
$ git clone https://github.com/ITDevLtd/blog_using_ftrace.git
```

Secondly, initialise and update git submodules:

```
$ git submodule init
$ git submodule update
```

The root directory is split into two main directories: one for the driver and one for a test application that will exercise the driver and read from the binary file it creates when it loads. The test application is *very* basic, as is the driver, so that we can concentrate on using `ftrace` .

To compile the driver, you just need access to an Ubuntu based machine. This should work just as well on other distributions but installing the kernel headers will probably be done differently[1].

Ensure you have the essential build tools:

```
$ sudo apt install build-essential
```

The first thing you need are the Linux kernel headers (and `libelf` debug libraries and header files). The reason for this is that modules must be compiled against the exact kernel headers for the kernel on which they will run. From the command line type the following:

```
$ sudo apt-get install linux-headers-$(uname -r) libelf-dev
```

Change into the driver directory and simply type:

```
$ make
```

You can install the compiled module using the `start.sh` script, which you must run with `sudo`. This is required as inserting a driver module requires elevated privileges.

```
$ sudo sh start.sh
```

The application code that is of interest for this blog is found in the file `blog_app.c`. Once you have compiled and loaded the driver module above, you can change to the app directory and build the application and the debug application:

```
$ make blog_app
$ make blog_app_debug
```

Manually Configure and Use the Function Tracer

At this point you should have compiled the device driver as a module and loaded it using the `start.sh` script as described. You should also have compiled the application.

Imagine you are the driver author about to run her test programme to ensure that data can be read back from the device via the binary `sysfs` file `/sys/devices/itdev/special_data`. She is expecting the two consecutive reads to complete immediately and output some data, but what she sees instead is this:

```
$ ./blog_app
Read "AABBCCDDEEFFGGHHIIJJKK"
Read ""
```

Not what she wanted! The same data block should have been read back twice.

As described earlier, a quick debug skirmish didn't reveal much for our author, so she decides to use `ftrace`. Before making any code changes, a look at the raw tracing output might reveal something.

The first thing to do is select the function tracer so that a peak into a trace of all the functions called within the kernel can be seen.

One thing to note is that the location of the `tracefs` files may vary depending on your system and kernel version. In the description below it is assumed that you're running on a kernel before 4.1 and have mounted `debugfs` in the usual place.

She must **run as root** and type the following to select the function graph tracer:

```
# echo "function_graph" > /sys/kernel/debug/tracing/current_tracer
```

Then she enables tracing, runs her programme and immediately disables tracing using the following commands:

```
# echo 1 > /sys/kernel/debug/tracing/tracing_on; \
  ./blog_app; \
  echo 0 > /sys/kernel/debug/tracing/tracing_on
```

Once the programme finishes, she can look at the trace log by dumping out the following file:

```
# cat /sys/kernel/debug/tracing/trace
```

This produces an absolute ton of textual output, so she decides to repeat the command and send it to a file so that she can examine it in her text editor.

```
# cat /sys/kernel/debug/tracing/trace > my_trace_output
```

At first glance the file is large and has traces from different CPUs all mixed up together. She searches for `itdev_example_cdev_read_special_data`, finds it and deletes all lines that do not have the same CPU number as this to reduce the trace size and remove "noise". This process has been embodied by running the script:

```
# ./ftrace_blogapp.sh 0 > my_trace_output
```

She now sees the following (your output may differ slightly):

```
...
1)               |  do_syscall_64() {
1)               |    SyS_read() {
...
1)               |        __vfs_read() {
1)               |          kernfs_fop_read() {
...
1)               |            mutex_lock() {
1)               |              _cond_resched() {
1)   0.041 us    |                rcu_all_qs();
1)   0.337 us    |              }
1)   0.635 us    |            }
1)   0.042 us    |            kernfs_get_active();
1)             |            sysfs_kf_bin_read() {
1)             |              itdev_example_cdev_read_special_data [most_basic]() {
...
1)               |        __vfs_read() {
...
1)               |            mutex_lock() {
1)               |              _cond_resched() {
1)   0.041 us    |                rcu_all_qs();
1)   0.339 us    |              }
1)   0.639 us    |            }
1)   0.045 us    |            kernfs_get_active();
1)  0.062 us   |            sysfs_kf_bin_read();
1)   0.064 us    |            kernfs_put_active();
1)   0.043 us    |            mutex_unlock();
1)   0.055 us    |            __check_object_size();
1)   0.094 us    |            kfree();
1)   4.361 us    |          }
1)   4.684 us    |        }
```

"Great", she thinks, "I can see that my `read()` call results in a tree of kernel function calls that eventually call my driver function `itdev_example_cdev_read_special_data()`. The parent function in the call graph is `sysfs_kf_bin_read()`. I can see that my driver function is called once but `sys_kf_bin_read()` is called twice, so the last call must be the one originating from my test that doesn't return any data to me. The answer to my problem must lie in `sysfs_kf_bin_read()`!".

Thus, she can quickly consult the Linux source for her kernel version. The easiest option, if you don't have a kernel checkout to hand, is just to use Bootlin's Elixir linux source cross reference tool (https://elixir.bootlin.com/linux/latest/source).

And the problem becomes immediately apparent. She looks at `sysfs_kf_bin_read()` in Github (https://github.com/torvalds/linux/blob/v5.3/fs/sysfs/file.c#L79) and sees:

```
loff_t size = file_inode(of->file)->i_size;
...
if (size) {
    if (pos >= size)
        return 0;
    if (pos + count > size)
        count = size - pos;
}
...
return battr->read(of->file, kobj, battr, buf, pos, count);
```

The file has an "`i_size`" attribute and when the offset into the file exceeds the size, nothing is returned. However, she observes the "`if (size)`" condition and immediately realises that if the file has an "`i_size`" of zero then it will always be read! "So", she thinks, "what size did I create this file within the driver?" and looks at the driver source code and sees the following:

```
sysfs_bin_attr_init(&gbl_ctx.battr);
gbl_ctx.battr.attr.name = "special_data";
...
gbl_ctx.battr.size = test_data_block_len;
```

And we have found the problem. The attribute `gbl_ctx.battr.size` is not zero. She changes the value to zero, recompiles and installs the driver, then re-tests. Now everything works as expected:

```
$ ./blog_app
Read "AABBCCDDEEFFGGHHIIJJKK"
Read "AABBCCDDEEFFGGHHIIJJKK"
```

With a little knowledge of the kernel internals, together with the help of `ftrace`, the author has been able to quickly identify her problem. She's a little annoyed with herself, because she knows the correct API usage for `sysfs_create_bin_file()`; it was just tough to see the error by mere code inspection.

Hone In: Use the Function Tracer From Within Your Application

Tracking down that little error during development was easy with `ftrace`. Our developer was quickly able to figure out the reason for the error and correct it.

The example presented here is simple. What if it were far more complex, running through a network of functions within her driver and the issue only occurred when large amounts of the system were being exercised during test? Using the above simple strategy where tracing is enabled for the entire test run now fails her. There is too much information being dumped out.

What she'd really like is some specific "markers" that can be grep'ed for in the trace dump to find the location of suspect call paths more easily and potentially examine variable values etc.

Injecting Trace Markers from the Kernel

To continue with our example code, our driver author might want to examine the value of some of the variables passed to the binary file's read function. Obviously, she already has her answer, but let's imagine the function is more complex and inspecting variable values becomes worthwhile.

"Trace markers" are user-defined messages that can be "injected" into the function trace. They appear as comments. Their largest advantage, apart from being embedded in the correct place in the reported function, is that, because they are buffered, they are potentially more efficient than a `printk()` . This is because `printk()` may have to print its output over a slow serial connection and will cause less disturbance of the timing of the kernel source under debug.

Our author decides to put such a marker in the driver's binary file read function, `itdev_example_cdev_read_special_data()` by adding this line:

```
trace_printk(TAG "Reading %zu bytes\n", size);
```

If she looked at the function trace *before* she'd corrected the error, she would see the following:

```
1)               |        __vfs_read() {
1)               |          kernfs_fop_read() {
...
1)               |            sysfs_kf_bin_read() {
1)               |              itdev_example_cdev_read_special_data [most_basic]() {
1)             |                /* ITDev: Reading 22 bytes */
```

You can see that `trace_printk()` accepts the usual printf format string. It prints this string into the function trace as a comment. This makes it easier to examine variables, and also to grep the trace, as in our case the macro TAG expanded to "ITDev: " so that all our `trace_printk()` messages were easy to find in the file.

Injecting Trace Markers from User Space

Let's say that things are more complex still. Perhaps the application makes many different reads of the binary file or there are many such files. Our author has been debugging her code and now thinks that the problem occurs when a specific portion of the test code executes. All the other trace noise generated has become inconsequential and slightly annoying and she wishes to "cut" the rest out.

In our application, all of the `ftrace` code has been hidden away in a few files, `itd_ftrace_*` and `file_io.c` so as not to obscure the application code. As independent files, the code that writes to the trace and enables/disables tracing, can be used with any project that needs a quick debug – hopefully a reusable code snippet that can help you on your way.

There is a huge amount that `ftrace` can do and only the tip of the iceberg is covered here and in the code snippets.

To further reduce the `ftrace` size and only trace within certain programme sections, our developer could augment the test code with some mechanisms that only enable tracing around the suspect areas of code, and nowhere else. This can be done in the same way as before, by writing to the relevant `tracefs` files, except rather than doing it manually from the command line it has to be done from within the test application. One drawback is that the application under test now must run with root privileges, however, this is only applicable during debug so should be safe enough. Once debugging is over, the `ftrace` calls must be removed.

If you refer to `blog_app.c` , you will see several calls to the functions `(un)init_debug_tracing()` , `itd_trace_on()` , `itd_trace_off()` and `itd_trace_print()` .

These functions package the control of the function tracer so as not to pollute the source code with a ton of debug code that would further obscure the reading of the programme's logic. You can use these functions in your own code, they're released under the GPLv2 license. They have a couple of advantages.

1. The API automatically figures out where the `tracefs` files are mounted. It deals with older kernels, where the trace related files are part of the `debugfs` and newer kernels where the trace related files are part of a separate trace filesystem (`tracefs`). It also copes with mounts of either of these that do not appear in the standard locations.
2. Because the API hides all the writing to the trace files that we did manually in the previous section, the software being tested does not get overly polluted with debug code that would otherwise make it harder to read and understand.

Our author has added a few of these calls to her application. When she compiled `./blog_app` these calls were linked to stub functions. When she compiles `./blog_app_debug` , these calls are linked to the real functions.

The debug version of the application can be run without having to "manually" modify the `tracefs` files from the command line. This is all done from within the application itself.

This process has been embodied by running the script " `sudo ./ftrace_blog_app 1` ". If you examine the output of the function trace produced, you will see where messages from the application itself have been injected into the trace. For more complex application debugs, this can prove invaluable.

Doing More

There are many more pseudo files in the `tracefs` filesystem. These can be used to configure the function trace in many ways. They are beyond the scope of this blog, but the reader is strongly encouraged to investigate.

Conclusion

We've seen how the function tracer within `ftrace` can be used to find out what's going on inside the kernel and therefore help with Linux driver development. The example above shows how a developer can inspect the driver and kernel call sequences to visualise the interaction between them. This is a very powerful built-in Linux utility.

I hope that you've found this article useful and that it has given you some small insight into other possible ways of debugging driver code and the kernel internals in general.

How ITDev Can Help

Many of our client projects involve embedded systems and several of these run Linux or Android as an operating system. We have extensive experience in building, modifying and debugging Linux kernels, including creating custom Linux builds optimised for the specific task. We have created demos for clients, where we have needed to modify the bootloader to use an updated Android kernel.

We are always happy to offer advice and assistance to companies considering or already using Linux or Android on their embedded system. If you have a need and are unsure what approach to take, get in contact. Our initial discussions are always free of charge, so if you have any questions, or would like to find out more, we would be delighted to speak to you.

To arrange an initial consultation email us (mailto:howcanwehelp@itdev.co.uk?subject=Enquiry) or call us on +44 (0)23 8098 8890.

- For future news and blog updates, follow us on Linkedin (https://www.linkedin.com/company/235042), Twitter (https://twitter.com/ITDevLtd) and Facebook (https://www.facebook.com/itdevltd/).

Embedded Linux workshop

In addition to supporting Linux kernel and driver development, through upstreaming patches and modifications, we are considering running an 'online' workshop. If you would be interested in attending a 'virtual' workshop, sign up to our Embedded Linux Interest Group (http://eepurl.com/dCms-P) to be kept informed.

Click to find out about our Embedded Linux Interest Group, including upcoming events

(http://eepurl.com/dCms-P)

Footnote

[1] For example, on Fedora (https://getfedora.org/), you would need to use:

```
$ sudo yum install "kernel-devel-uname-r == $(uname -r)"
$ sudo dnf install elfutils-libelf-devel
```

Proud members of nmi Semiconductors to Systems WE'RE CONNECTED (http://www.nmi.org.uk/)

IET The Institution of Engineering and Technology Enterprise Partner (https://www.theiet.org/)