Fprobe - Function entry/exit probe

Introduction

Fprobe is a function entry/exit probe mechanism based on ftrace. Instead of using ftrace full feature, if you only want to attach callbacks on function entry and exit, similar to the kprobes and kretprobes, you can use fprobe. Compared with kprobes and kretprobes, fprobe gives faster instrumentation for multiple functions with single handler. This document describes how to use fprobe.

The usage of fprobe

The fprobe is a wrapper of ftrace (+ kretprobe-like return callback) to attach callbacks to multiple function entry and exit. User needs to set up the *struct fprobe* and pass it to *register_fprobe()*.

Typically, *fprobe* data structure is initialized with the *entry_handler* and/or *exit_handler* as below.

```
struct fprobe fp = {
       .entry_handler  = my_entry_callback,
       .exit_handler   = my_exit_callback,
};
```

To enable the fprobe, call one of register_fprobe(), register_fprobe_ips(), and register_fprobe_syms(). These functions register the fprobe with different types of parameters.

The register_fprobe() enables a fprobe by function-name filters. E.g. this enables @fp on "func*()" function except "func2()".:

```
register_fprobe(&fp, "func*", "func2");
```

The register_fprobe_ips() enables a fprobe by ftrace-location addresses. E.g.

```
unsigned long ips[] = { 0x.... };

register_fprobe_ips(&fp, ips, ARRAY_SIZE(ips));
```

And the register_fprobe_syms() enables a fprobe by symbol names. E.g.

```
char syms[] = {"func1", "func2", "func3"};

register_fprobe_syms(&fp, syms, ARRAY_SIZE(syms));
```

To disable (remove from functions) this fprobe, call:

```
unregister_fprobe(&fp);
```

You can temporally (soft) disable the fprobe by:

```
disable_fprobe(&fp);
```

and resume by:

```
enable_fprobe(&fp);
```

The above is defined by including the header:

```
#include <linux/fprobe.h>
```

Same as ftrace, the registered callbacks will start being called some time after the register_fprobe() is called and before it returns. See :file:`Documentation/trace/ftrace.rst`.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\trace\(linux-master)(Documentation)(trace)fprobe.rst, line 79); *backlink***
>
> Unknown interpreted text role "file".

Also, the unregister_fprobe() will guarantee that the both enter and exit handlers are no longer being called by functions after unregister_fprobe() returns as same as unregister_ftrace_function().

The fprobe entry/exit handler

The prototype of the entry/exit callback function is as follows:

```
void callback_func(struct fprobe *fp, unsigned long entry_ip, struct pt_regs *regs);
```

Note that both entry and exit callbacks have same ptototype. The @entry_ip is saved at function entry and passed to exit handler.

@fp
: This is the address of *fprobe* data structure related to this handler. You can embed the *fprobe* to your data structure and get it by container_of() macro from @fp. The @fp must not be NULL.

@entry_ip
: This is the ftrace address of the traced function (both entry and exit). Note that this may not be the actual entry address of the function but the address where the ftrace is instrumented.

@regs
: This is the *pt_regs* data structure at the entry and exit. Note that the instruction pointer of @regs may be different from the @entry_ip in the entry_handler. If you need traced instruction pointer, you need to use @entry_ip. On the other hand, in the exit_handler, the instruction pointer of @regs is set to the currect return address.

Share the callbacks with kprobes

Since the recursion safeness of the fprobe (and ftrace) is a bit different from the kprobes, this may cause an issue if user wants to run the same code from the fprobe and the kprobes.

Kprobes has per-cpu 'current_kprobe' variable which protects the kprobe handler from recursion in all cases. On the other hand, fprobe uses only ftrace_test_recursion_trylock(). This allows interrupt context to call another (or same) fprobe while the fprobe user handler is running.

This is not a matter if the common callback code has its own recursion detection, or it can handle the recursion in the different contexts (normal/interrupt/NMI.) But if it relies on the 'current_kprobe' recursion lock, it has to check kprobe_running() and use kprobe_busy_*() APIs.

Fprobe has FPROBE_FL_KPROBE_SHARED flag to do this. If your common callback code will be shared with kprobes, please set FPROBE_FL_KPROBE_SHARED *before* registering the fprobe, like:

```
fprobe.flags = FPROBE_FL_KPROBE_SHARED;

register_fprobe(&fprobe, "func*", NULL);
```

This will protect your common callback from the nested call.

The missed counter

The *fprobe* data structure has *fprobe::nmissed* counter field as same as kprobes. This counter counts up when;

- fprobe fails to take ftrace_recursion lock. This usually means that a function which is traced by other ftrace users is called from the entry_handler.
- fprobe fails to setup the function exit because of the shortage of rethook (the shadow stack for hooking the function return.)

The *fprobe::nmissed* field counts up in both cases. Therefore, the former skips both of entry and exit callback and the latter skips the exit callback, but in both case the counter will increase by 1.

Note that if you set the FTRACE_OPS_FL_RECURSION and/or FTRACE_OPS_FL_RCU to *fprobe::ops::flags* (ftrace_ops::flags) when registering the fprobe, this counter may not work correctly, because ftrace skips the fprobe function which increase the counter.

Functions and structures

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\trace\(linux-master)(Documentation)(trace)fprobe.rst, line 172)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/fprobe.h
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\trace\(linux-master)(Documentation)(trace)fprobe.rst, line 173)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: kernel/trace/fprobe.c
```