# Kernel Korner

## *Kprobes—a Kernel Debugger*

**R. Krishnakumar**

**Abstract**

Looking for a way to use some of the same debugging techniques in the kernel that you apply in user-space code? Here's how to bring debugging support to tricky kernel development problems.

---

Kprobes is a mechanism used to register breakpoints and corresponding handlers. After enabling Kprobes support in the kernel, we can debug any instruction at any kernel address. This article explains how to compile a kernel with Kprobes and how to register and unregister Kprobes, using a live example. It also covers the concept of debugging the kernel, plus internal operations of the Kprobes framework and its features.

To get started, suppose we are trying to debug a specific instruction at an address location in the kernel. Using the facilities provided by Kprobes, we can execute three functions, namely, pre-handler, post-handler and fault handler. The pre-handler function is executed before the execution of the instruction at the debugged memory location takes place. The post-handler executes after the instruction being debugged is executed. The fault handler is executed if the instruction leads to a fault.

To explain further, let's look at an example. Suppose we want to debug the instruction at location *x*. Let the instruction at location *x* be *i*. The function to be executed before *i* is executed, the pre-handler, is named pre_x. The function to be executed after the *i* is executed, the post-handler, is named post_x. The fault handler itself is fault_x.

Before *i* is executed, Kprobes runs the pre_x function. In the pre_x function we can do some necessary debugging actions, such as checking the contents of various registers and manipulating the registers. After the pre_x finishes executing, *i* executes, followed by post_x. The fault handler comes into the picture when the instruction *i* causes an operating system fault. If the fault occurs due to the execution of *i*, the fault handler, fault_x, is called.

# Features

A debugging console is not necessary when using Kprobes. This is a significant design point, because it results in minimal system dependencies for operation. It therefore allows debugging to be performed at interrupt time, during context switches, when the system is disabled for interrupts and so on.

In addition, no forced serialisation of system processes is required for operation. In particular, in an SMP environment no interprocessor serialisation is required.

Another important feature of Kprobes is that data can be extracted by a probe handler and saved in a buffer. This is significant for later examination of data from a crashdump or data dumped to the console at a consistent time.

# How to Enable Kprobes Support in the Kernel

After being out-of-tree patches for a long time, Kprobes finally was included in the vanilla Linux kernel. This article covers the core Kprobes functionality included as of kernel version 2.6.9. Many other features are supported by Kprobes, and they are available as patches from the Kprobes Web site (see the on-line Resources).

Download the vanilla kernel from http://www.kernel.org. While configuring the kernel, go to the Kernel Hacking submenu. Enable Kernel debugging, and then choose the Kprobes option. Compile the kernel with this configuration and boot it.

After we have enabled Kprobes, we can use various kernel APIs to register and unregister it. The function used to register Kprobe is register_kprobe. This function takes the pointer to a structure called struct kprobe. The definition of the structure is:

```
struct kprobe {
        struct hlist_node hlist;
        kprobe_opcode_t *addr;
        kprobe_pre_handler_t pre_handler;
        kprobe_post_handler_t post_handler;
        kprobe_fault_handler_t fault_handler;
        kprobe_break_handler_t break_handler;
        kprobe_opcode_t opcode;
        kprobe_opcode_t insn[MAX_INSN_SIZE];
};
```

In the struct we can specify the following:

1. The address on which Kprobe has to be set (addr).
2. The pre-handler to be executed (pre_handler).
3. The post-handler to be executed (post_handler).
4. The fault handler to be executed (fault_handler).

To unregister Kprobe, you can use unregister_kprobe, which takes the same argument as register_kprobe.

The prototype of register_kprobe and unregister_kprobe is simple:

```
int register_kprobe(struct kprobe *p);
void unregister_kprobe(struct kprobe *p);
```

You can find these definitions in include/linux/kprobes.h.

# Live Action

Let's look at a real example of the process of kernel debugging using Kprobes. We begin by inserting the function we are going to debug. The code to do this is as follows, I have added the line numbers for reference:

```
 1 /* Filename: first.c */
 2
 3 #include <linux/module.h>
 4 #include <linux/init.h>
```

```
 5
 6 int hello_to_debug(void)
 7 {
 8         printk("\nFrom the function - %s\n",
 9                                 __FUNCTION__);
10         return 0;
11 }
12
13 static void exit_to_debug(void)
14 {
15         printk("\nModule exiting \n");
16 }
17
18 static int init_to_debug(void)
19 {
20         printk("\nKeeping the function to debug"
21                "\nat the kernel address %p\n",
22                hello_to_debug);
23         return 0;
24 }
25
26 EXPORT_SYMBOL(hello_to_debug);
27 module_init(init_to_debug);
28 module_exit(exit_to_debug);
29
30 MODULE_AUTHOR ("Krishnakumar. R,
31                 <rkrishnakumar@gmail.com>");
32 MODULE_DESCRIPTION ("Kprobes test module");
33 MODULE_LICENSE("GPL");
```

Suppose we need to debug the function given in line 6, hello_to_debug. Begin by compiling the above code and insert it as a module. The EXPORT_SYMBOL directive at line 26 makes sure that the rest of the kernel code can see this function.

Now, insert Kprobe at the location to be debugged, the function hello_to_debug:

```
 1 /* Filename: kprobes.c */
 2
 3 #include <linux/module.h>
 4 #include <linux/init.h>
 5 #include <linux/kprobes.h>
 6
 7 static struct kprobe kpr;
 8 extern int hello_to_debug(void);
 9
10 static void __exit exit_probe(void)
11 {
12        printk("\nModule exiting \n");
13        unregister_kprobe(&kpr);
14 }
15
16 static int before_hook(struct kprobe *kpr,
17                        struct pt_regs *p)
18 {
19        printk("\nBefore hook");
20        printk("\nThis is the Kprobe pre \n"
21               "handler for instruction at \n"
22               "%p\n", kpr->addr);
23        printk("The registers are:\n");
24        printk("eax=%lx, ebx=%lx, ecx=%lx, \n"
25               "edx=%lx\n", p->eax,  p->ebx,
26               p->ecx,  p->edx);
27        printk("eflags=%lx, esp=%lx\n",
28                p->eflags,  p->esp);
```

```
29        return 0;
30 }
31
32 static int after_hook(struct kprobe *kpr,
33                       struct pt_regs *p,
34                       unsigned long flags)
35 {
36        printk("\nAfter hook");
37        printk("\nThis is the Kprobe post \n"
38               "handler for instruction at"
39               " %p\n", kpr->addr);
40        printk("The registers are:\n");
41        printk("eax=%lx, ebx=%lx, ecx=%lx, \n"
42               "edx=%lx\n", p->eax,  p->ebx,
43               p->ecx,  p->edx);
44        printk("eflags=%lx, esp=%lx\n",
45                p->eflags,  p->esp);
46        return 0;
47 }
48
49 static int __init init_probe(void)
50 {
51        printk("\nInserting the kprobes \n");
52        /* Registering a kprobe */
53        kpr.pre_handler =
54            (kprobe_pre_handler_t)before_hook;
55        kpr.post_handler =
56            (kprobe_post_handler_t)after_hook;
57        kpr.addr =
58           (kprobe_opcode_t *)(&hello_to_debug);
59        printk("\nAddress where the kprobe is \n"
60               "going to be inserted - %p\n",
61               kpr.addr);
62        register_kprobe(&kpr);
63        return 0;
64 }
65
66 module_init(init_probe);
67 module_exit(exit_probe);
68
69 MODULE_AUTHOR ("Krishnakumar. R,
70                 <rkrishnakumar@gmail.com>");
71 MODULE_DESCRIPTION ("Kprobes test module");
72 MODULE_LICENSE("GPL");
```

Line 57 specifies the address location where Kprobe should be set. Lines 53 and 55 specify the pre-handler and the post-handler functions, which should be activated corresponding to the address location. Line 62 registers Kprobe. So, when the above code is compiled and inserted as a module, Kprobe is registered at the hello_to_debug function. When the module is unloaded, Kprobe is unregistered, as shown in line 13.

Now we have to invoke the function we are debugging. This is done with the following code:

```
 1 /* Filename: call.c */
 2
 3 #include <linux/module.h>
 4 #include <linux/init.h>
 5
 6 extern int hello_to_debug(void);
 7
 8 static void __exit exit_to_debug(void)
 9 {
10         printk("\nModule exiting \n");
11 }
```

```
12
13 static int __init init_to_debug(void)
14 {
15         printk("\nCalling the function \n");
16         hello_to_debug();
17         return 0;
18 }
19
20 module_init(init_to_debug);
21 module_exit(exit_to_debug);
22
23 MODULE_AUTHOR ("Krishnakumar. R,
24                 <rkrishnakumar@gmail.com>");
25 MODULE_DESCRIPTION ("Kprobes test module");
26 MODULE_LICENSE("GPL");
```

Line 16 here calls the function we are debugging. The Kprobes framework invokes the pre-handler prior to the execution of the function, and the post-handler is invoked after the execution of the instruction under debug. We then can print the register contents and Kprobe information. The following is the transcript of messages I received after compiling and inserting the above modules.

Inserting the first module:

```
[root@kk code]# /sbin/insmod first.ko

Keeping the function to debug
at the kernel address c883a000
```

Inserting the Kprobes placing module:

```
[root@kk code]# /sbin/insmod kprobes.ko

Inserting the kprobes

Address where the kprobe is
going to be inserted - c883a000
```

Calling the function under debug:

```
[root@kk code]# /sbin/insmod call.ko

Calling the function

Before hook
This is the Kprobe pre
handler for instruction at
c883a000
The registers are:
eax=17, ebx=c47ba000, ecx=c1264090,
edx=c47ba000
eflags=296, esp=c884000f

After hook
This is the Kprobe post
handler for instruction at c883a000
The registers are:
eax=17, ebx=c47ba000, ecx=c1264090,
edx=c47ba000
eflags=196, esp=c883a09e

From the function - hello_to_debug
```

# Breakpoints and Debuggers

To understand better how Kprobes works, we should know the general concept of breakpoints, because Kprobes makes use of the same mechanism. A breakpoint is a mechanism provided by the hardware in most processors that we can use for debugging. For now, we are going to consider the x86 architecture. The instruction set for the processor provides a breakpoint instruction, and this instruction generates a breakpoint exception. Thus, control is transferred to the breakpoint exception handler. Most debuggers use this facility.

Suppose the debugger makes use of the breakpoint mechanism to debug. If it has to debug an instruction at a particular location, it replaces the corresponding instruction with the breakpoint instruction. The breakpoint instruction then generates the exception. The debugger contains a provision to be informed whenever such an exception is generated. The debugger then takes the necessary debugging steps, such as printing out the register values and manipulating them, as well as replacing the instruction with the original instruction. After this, execution of the instruction proceeds as usual.

# Pre-handler

When we register a pre-handler, what actually happens is Kprobes replaces the instruction at the memory location with a breakpoint instruction. The instruction that was present there is saved for later reference.

The following lines from the function int register_kprobe(struct kprobe *p) in the kernel/kprobes.c do this:

```
p->opcode = *p->addr;
*p->addr = BREAKPOINT_INSTRUCTION;
```

Hence, whenever control reaches the particular location, the breakpoint exception occurs. The default breakpoint exception handler is modified by Kprobes. The modified exception handler checks whether the address has an instance of Kprobe associated with it. If there is an associated Kprobe, the exception handler executes the pre-handler. Otherwise, control is transferred to the normal breakpoint exception handler. If Kprobe is registered for that particular location, it prepares the processor to call the post-handler, which takes over once the pre-handler has executed.

The function responsible for handling the breakpoint is listed below:

```
asmlinkage int do_int3(struct pt_regs *regs,
                       long error_code);
```

and the function that invokes the pre-handler is here:

```
static inline int kprobe_handler(struct pt_regs *regs);
```

# Post-Handler

The post-handler is executed after the instruction with which we associate the probe has executed. To facilitate this, the Kprobes framework gets some help from the hardware, specifically from a processor feature called trap generation.

If we have set the trap flag of the processor, it generates a trap exception after every instruction. After the pre-handler is run, the Kprobes framework sets the trap flag. It then replaces the breakpoint instruction with the original instruction. The function that prepares for the post-handler is presented below:

```
static inline void prepare_singlestep(struct kprobe *p,
            struct pt_regs *regs);
```

After the instruction we are debugging has executed, the processor generates a trap exception. The function responsible for the exception handling of the trap generation looks like this:

```
asmlinkage void do_debug(struct pt_regs * regs,
                         long error_code);
```

and the function that does the necessary activities for the Kprobes post-handler is:

```
static inline int post_kprobe_handler(struct pt_regs *regs);
```

The post_kprobe_handler function calls the post-handler that we have registered for that particular probe.

# Fault Handler

The fault handler is executed whenever a fault is generated when executing the instruction under debug. The function responsible for Kprobes' activities on faults looks like this:

```
static inline int kprobe_fault_handler(struct pt_regs *regs,
                          int trapnr);
```

This function is called under two circumstances:

1. Whenever a general protection fault occurs, do_general_protection, and we know that it has been generated by a Kprobes instruction.

2. Whenever a device-not-available fault generation occurs, and we know it has been generated by a Kprobes instruction.

In either of these cases, the fault handler can be used to discover what went wrong.

# Conclusion

The Kprobes patch helps a kernel developer debug any address within the kernel. Various patches are available from the Kprobes home page, including ones for setting watch points and for debugging user address locations. With proper use, Kprobes can become a powerful weapon in any kernel developer's arsenal.

# Acknowledgements

I am thankful to Richard J Moore and Andrew Morton for their valuable comments on the draft version of this article, to Manish P Fadnavis for his support and to Pramode C E, Shine Mohammed Jabbar and Chitkala Sethuraman for their feedback.

**Resources for this article:** http://www.linuxjournal.com/article/8136.