Kernel debugging with Kprobes

Insert printk's into the Linux kernel on the fly

Level: Intermediate

Prasanna Panchamukhi (prasanna@in.ibm.com), Developer, Linux Technology Center, IBM India Software Labs

19 Aug 2004

Collecting debugging information from the Linux™ kernel using printk is a well-known method -- and with Kprobes, it can be done without the need to constantly reboot and rebuild the kernel. Kprobes, in combination with 2.6 kernels, provides a lightweight, non-disruptive, and powerful mechanism to insert printk's dynamically. Logging debug info, such as the kernel stack trace, kernel data structures, and registers, has never been so easy!

Kprobes is a simple and lightweight mechanism in Linux that allows you to insert breakpoints into a running kernel. Kprobes provides an interface to break into any kernel routine and collect information non-disruptively from the interrupt handler. Debugging information, such as processor registers and global data structures, can be easily collected using Kprobes. Developers can even use Kprobes to modify register values and global data structure values.

To accomplish this, Kprobes inserts a probe by dynamically writing breakpoint instructions at a given address in the running kernel. Execution of the probed instruction results in a breakpoint fault. Kprobes hooks in to the breakpoint handler and collects the debugging information. Kprobes can even single-step probed instructions.

Installation

To install Kprobes, download the latest patch from the Kprobes home page (see Resources for a link). The tarred file will be named something along the lines of kprobes-2.6.8-rc1.tar.gz. Untar the patch and apply it to the Linux kernel:

```
$tar -xvzf kprobes-2.6.8-rc1.tar.gz
$cd /usr/src/linux-2.6.8-rc1
$patch -p1 < ../kprobes-2.6.8-rc1-base.patch
```

Kprobes makes use of the **SysRq** key, an artifact from the days of DOS that has found many new uses under Linux (see Resources). You'll find the **SysRq** key to the left of the **Scroll Lock** key; it's often also labeled **Print Screen**. To enable the **SysRq** key for Kprobes, apply the kprobes-2.6.8-rc1-sysrq.patch patch:

```
$patch -p1 < ../kprobes-2.6.8-rc1-sysrq.patch
```

Configure the kernel with `make xconfig/ make menuconfig/ make oldconfig` and enable `CONFIG_KPROBES` and `CONFIG_MAGIC_SYSRQ` flags. Build and boot into the new kernel. You are now ready to insert printk's and collect debugging information dynamically and unobtrusively by writing simple Kprobes modules.

Writing Kprobes modules

For each probe, you will need to allocate the structure `struct kprobe kp;` (see include/linux/kprobes.h for more information on this).

Listing 1. Defining pre, post, and fault handlers

```
 /* pre_handler: this is called just before the probed instruction is
  *     executed.
  */

int handler_pre(struct kprobe *p, struct pt_regs *regs) {
        printk("pre_handler: p->addr=0x%p, eflags=0x%lx\n",p->addr,
                regs->eflags);
        return 0;
}

 /* post_handler: this is called after the probed instruction is executed
  *     (provided no exception is generated).
  */

void handler_post(struct kprobe *p, struct pt_regs *regs, unsigned long flags) {
        printk("post_handler: p->addr=0x%p, eflags=0x%lx \n", p->addr,
                regs->eflags);
}

 /* fault_handler: this is called if an exception is generated for any
  *     instruction within the fault-handler, or when Kprobes
  *     single-steps the probed instruction.
  */

int handler_fault(struct kprobe *p, struct pt_regs *regs, int trapnr) {
        printk("fault_handler:p->addr=0x%p, eflags=0x%lx\n", p->addr,
                regs->eflags);
        return 0;
}
```

Getting the address of a kernel routine

You also need to specify the address of the kernel routine where you want to insert the probe during registration. Use any of these methods to get the kernel routine address:

1. **Get the address directly from the System.map file.**
   For example, to get the address of do_fork, execute `$grep do_fork /usr/src/linux/System.map` at the command line.
2. **Use the `nm` command.**
   `$nm vmlinuz |grep do_fork`
3. **Obtain the address from the /proc/kallsyms file.**
   `$cat /proc/kallsyms |grep do_fork`
4. **Use the `kallsyms_lookup_name()` routine.**
   This routine is defined in the kernel/kallsyms.c file, and you must compile the kernel with `CONFIG_KALLSYMS` enabled in order to use it. `kallsyms_lookup_name()` takes a kernel routine name as a string and returns the address of that kernel routine. For example: `kallsyms_lookup_name("do_fork");`

Then register your probe in the init_module:

Listing 2. Registering a probe

```
 /* specify pre_handler address
  */
       kp.pre_handler=handler_pre;
 /* specify post_handler address
  */
       kp.post_handler=handler_post;
 /* specify fault_handler address
  */
       kp.fault_handler=handler_fault;
 /* specify the address/offset where you want to insert probe.
  * You can get the address using one of the methods described above.
  */
       kp.addr = (kprobe_opcode_t *) kallsyms_lookup_name("do_fork");

 /* check if the kallsyms_lookup_name() returned the correct value.
  */
       if (kp.add == NULL) {
               printk("kallsyms_lookup_name could not find address
                                       for the specified symbol name\n");
               return 1;
       }

 /*     or specify address directly.
  * $grep "do_fork" /usr/src/linux/System.map
  * or
  * $cat /proc/kallsyms |grep do_fork
  * or
  * $nm vmlinuz |grep do_fork
  */
       kp.addr = (kprobe_opcode_t *) 0xc01441d0;

 /* All set to register with Kprobes
  */
       register_kprobe(&kp);
```

Once the probe is registered, running any shell command will result in a call to `do_fork`, and you will be able to see your printk's on the console, or by running `dmesg`. Remember to unregister the probe when you are done:

```
unregister_kprobe(&kp);
```

The following output shows kprobe's address, and the contents of the eflags registers:

```
$tail -5 /var/log/messages

Jun 14 18:21:18 llm05 kernel: pre_handler: p->addr=0xc01441d0, eflags=0x202
Jun 14 18:21:18 llm05 kernel: post_handler: p->addr=0xc01441d0, eflags=0x196
```

Getting the offset

You can insert printk's at the beginning of a routine or at any offset in the function (the offset must be at the instruction boundary). The following code samples show how to calculate the offset. First, disassemble the machine instructions from the object file and save them as a file:

```
$objdump -D /usr/src/linux/kernel/fork.o > fork.dis
```

Which produces:

Listing 3. Disassembled fork

```
000022b0 <do_fork>:
    22b0:       55                      push   %ebp
    22b1:       89 e5                   mov    %esp,%ebp
    22b3:       57                      push   %edi
    22b4:       89 c7                   mov    %eax,%edi
    22b6:       56                      push   %esi
    22b7:       89 d6                   mov    %edx,%esi
    22b9:       53                      push   %ebx
    22ba:       83 ec 38                sub    $0x38,%esp
    22bd:       c7 45 d0 00 00 00 00    movl   $0x0,0xffffffd0(%ebp)
    22c4:       89 cb                   mov    %ecx,%ebx
    22c6:       89 44 24 04             mov    %eax,0x4(%esp)
    22ca:       c7 04 24 0a 00 00 00    movl   $0xa,(%esp)
    22d1:       e8 fc ff ff ff          call   22d2 <do_fork+0x22>
    22d6:       b8 00 e0 ff ff          mov    $0xffffe000,%eax
    22db:       21 e0                   and    %esp,%eax
    22dd:       8b 00                   mov    (%eax),%eax
```

To insert the probe at offset 0x22c4, get the relative offset from the beginning of the routine `0x22c4 - 0x22b0 = 0x14` and then add the offset to the address of do_fork `0xc01441d0 + 0x14`. (To ascertain the address of do_fork, run `$cat /proc/kallsyms | grep do_fork`.)

You can also add the relative offset of do_fork `0x22c4 - 0x22b0 = 0x14` to the output of `kallsyms_lookup_name("do_fork");` Thus: `0x14 + kallsyms_lookup_name("do_fork");`

Dumping kernel data structures

Now, let's dump some elements of all of the jobs that are running on the system with a Kprobe post_handler that we've modified to dump data structures:

Listing 4. Modified Kprope post_handler to dump data structures

```
void handler_post(struct kprobe *p, struct pt_regs *regs, unsigned long flags) {
        struct task_struct *task;
        read_lock(&tasklist_lock);
        for_each_process(task) {
                printk("pid =%x task-info_ptr=%lx\n", task->pid,
                        task->thread_info);
                printk("thread-info element status=%lx,flags=%lx, cpu=%lx\n",
                        task->thread_info->status, task->thread_info->flags,
                        task->thread_info->cpu);
        }
        read_unlock(&tasklist_lock);
}
```

This module should be inserted at the offset of do_fork.

Listing 5. Output of struct thread_info for pids 1508 and 1509

```
$tail -10 /var/log/messages

Jun 22 18:14:25 llm05 kernel: thread-info element status=0,flags=0, cpu=1
Jun 22 18:14:25 llm05 kernel: pid =5e4 task-info_ptr=f5948000
Jun 22 18:14:25 llm05 kernel: thread-info element status=0,flags=8, cpu=0
Jun 22 18:14:25 llm05 kernel: pid =5e5 task-info_ptr=f5eca000
```

Enabling the magic SysRq key

We already compiled in support for the SysRq key. Enable it with:

```
$echo 1 > /proc/sys/kernel/sysrq
```

Now you can use **Alt+SysRq+W** to view all inserted kernel probes on the console, or in /var/log/messages.

Listing 6. /var/log/messages shows a Kprobe inserted at do_fork

```
Jun 23 10:24:48 linux-udp4749545uds kernel: SysRq : Show kprobes
Jun 23 10:24:48 linux-udp4749545uds kernel:
Jun 23 10:24:48 linux-udp4749545uds kernel: [<c011ea60>] do_fork+0x0/0x1de
```

Better debugging with Kprobes

Because probe event handlers run as extensions to the system breakpoint interrupt handler, they have little or no dependence on system facilities -- and so are able to be implanted in the most hostile environments, from interrupt-time, and task-time, to disabled, inter-context switch, and SMP-enabled code paths -- all without adversely skewing system performance.

The benefits of using Kprobes are many. printk's can be inserted without rebuilding and rebooting the kernel. Processor registers can be logged and even modified for debugging -- without disruption to the system. Similarly, Linux kernel data structures can also be logged and even modified non-disruptively, as well. You can even debug race conditions on SMP systems with Kprobes -- and save yourself the trouble of all that rebuilding and rebooting. You'll find kernel debugging is faster and easier than ever.

Resources

- Find more information, recent news, READMEs, and downloads at the Kprobes home page. The README describes the kprobes interface in detail.
- Kprobes was developed from the full Dynamic Probes patch. Dynamic Probes use Kernel Hooks to gather difficult-to-acquire diagnostic information.
- Support for Kprobes was merged into the kernel at v.2.5.26; see Support for kernel probes (*Kernel Traffic*, July 25 2002) for the announcement and a brief write-up. KProbes added out-of-line single-stepping in 2.5.73.
- Kprobes makes use of the BIOS interrupt **SysRq** key. This can be made into a Magic SysRq key to defeat spyware, uncleanly reboot, show memory information, kill processes, and more (much more). It's good for debugging; less good for production machines, where it can pose a security threat.
- objdump displays information about one or more object files. For more information, see the objdump man page Linux 2.6 kernel modules.
- Captain's Universe has posted a document on HOWTO compile kernel modules for the kernel 2.6.
- Download the source of the kernel Module Utilities for 2.6; this replaces modutils for modern kernels. It's one of Rusty Russell's many useful patches.
- Find more resources for Linux developers in the developerWorks Linux zone.
- Browse for books on these and other technical topics.
- Develop and test your Linux applications using the latest IBM tools and middleware with a developerWorks Subscription: you get IBM software from WebSphere, DB2, Lotus, Rational, and Tivoli, and a license to use the software for 12 months, all for less money than you might think.
- Download no-charge trial versions of selected developerWorks Subscription products that run on Linux, including WebSphere Studio Site Developer, WebSphere SDK for Web services, WebSphere Application Server, DB2 Universal Database Personal Developers Edition, Tivoli Access Manager, and Lotus Domino Server, from the Speed-start your Linux app section of developerWorks. For an even speedier start, help yourself to a product-by-product collection of

how-to articles and tech support.

About the author

Prasanna S. Panchamukhi works as a developer for IBM's Linux Technology Center in Bangalore, India. He is currently involved in improving various debugging tools for Linux. Previously, he was involved in writing fiber channel device drivers and developing network processor applications, as well as maintaining Unix operating systems. You can reach Prasanna at prasanna@in.ibm.com.