【linux-2.6.31】Kernel Probes (Kprobes)

Authors : Jim Keniston <jkenisto@us.ibm.com>
: Prasanna S Panchamukhi <prasanna@in.ibm.com>

Translated By : openspace

Date : 2009-12-30

目　录

1. 概念：Kprobes、Jprobes、Return Probes

使用 Kprobes 可以动态进入任意的内核例程内部，并收集调试和性能信息，而这些操作不会破坏系统的运行。使用 Kprobes 几乎可以对内核代码中的任意地址进行 trap，指定的处理例程可以在到达断点时被调用。

当前有三种探测类型：kprobes、jprobes 和 kretprobes(也称为 return probes)。可以向内核中大多数指令插入一个 kprobe；可以在内核函数的入口插入 jprobe，jprobe 方便了对函数参数的访问；当一个指定的函数返回时触发 return probe。

通常，基于 Kprobes 的测试功能打包成一个内核模块。模块的 init 函数安装（“注册”）一个或多个 probe，exit 函数负责将它们注销。注册函数例如 register_kprobe()指定了 probe 插入的位置以及当触发 probe 时要调用的处理函数。

register_/unregister_*probes()函数可以对一组*probes 进行批处理注册/注销。当需要一次注销多个 probe 时，这些函数可以加快注销过程。

下面 3 个子章节描述了这些不同类型的 probe 是如何工作的，包括使用 Kprobes 需要知道的一些信息——例如，pre_hander 和 post_handler 的不同、如何利用 kretprobe 的 maxactive 和 nmissed 字段。如果急于使用 Kprobes，可以跳过直接进入第 2 部分。

1.1 Kprobe 如何工作

注册一个 kprobe 时，Kprobes 复制要探测的指令，并将其最初的一个或多个字节替换为一个断点指令（例如，在 i386 和 x86_64 上的 int3）。

CPU 运行到断点指令时，发生 trap，CPU 的寄存器信息被保存，控制权通过 notifier_call_chain 机制传给 Kprobes。Kprobes 执行与 kprobe 关联的"pre_handler"，并将 kprobe 结构的处理函数的地址和保存的寄存器内容传递给该函数。

接下来，Kprobes 单步执行复制的要探测的指令。（原地单步执行实际的指令更方便些，但是那样的话 Kprobes 需要临时移除断点指令。这会开启一个小的时间窗口，而其他 CPU 可能会直接通过探测点。）

单步执行完指令后，Kprobes 执行与 kprobe 关联的"post_handler"，如果有的话。然后继续执行探测点后面的指令。

1.2 Jprobe 如何工作

jprobe 的实现基于设置于一个函数的入口点上的 kprobe。它利用一种简单的反射机制提供对函数参数的无缝访问。jprobe 处理例程的签名（参数列表和返回类型）要与被探测的函数相同，而且函数最后需要调用 Kprobes 函数 jprobe_return()。

这里描述 Jprobe 是如何工作的。当触发探测时，Kprobes 复制保存的寄存器内容和栈的一大部分内容（参考下面）。然后 Kprobes 指出 jprobe 的处理函数处保存的指令指针，并从 trap 返回。结果，控制权转交给处理函数，处理函数看起来与被探测的函数有着相同的寄存器和栈内容。执行完成以后，处理函数调用 jprobe_return()，该函数会再产生一次 trap，恢复原始栈内容和处理器状态信息，并切换到被探测的函数。

通常，被调用函数有自己的参数，这样 gcc 生成的代码可以修改栈中这部分数据。

所以 Kprobes 需要保存栈的副本，并在 jprobe 处理函数运行返回之后恢复。最多可以复制 MAX_STACK_SIZE 字节——例如，在 i386 上是 64 字节。

注意被探测函数的参数可以通过栈或者寄存器传递。两种情况下 jprobe 都可以工作，只要处理函数的原型和被探测函数的一致就可以。

1.3 Return Probes

1.3.1 Return Probe 如何工作

调用 register_kretprobe()时，Kprobes 构造一个函数入口点的 kprobe。当调用被探测的函数时，该探测被触发，Kprobes 保存返回地址的副本，然后将返回地址替换为一个"trampoline"的地址。trampoline 是任意一段代码——通常为一条 nop 指令。引导时，Kprobes 向 trampoline 注册一个 kprobe。

被探测函数执行返回指令时，控制权转交给 trampoline 并触发探测。Kprobes 的 trampoline 处理函数调用用户指定的与 kretprobe 关联的返回处理函数，然后将保存的指令指针设置为保存的返回地址，这样从 trap 返回后可以继续执行。

执行被探测的函数时，它的返回地址保存在一个 kretprobe_instance 类型的对象中。在调用 register_kretprobe()之前，用户设置 kretprobe 结构的 maxactive 字段以指定可以同时探测的指定函数实例的数量。register_kretprobe()会预先分配指定数量的 kretprobe_instance 对象。

例如，如果函数是非递归的，并且调用时持有自旋锁，那么 maxactive 应该为 1。如果函数是非递归的并且从不放手 CPU（例如通过一个信号量或者抢占），那么 NR_CPUS 就可以了。如果 maxactive <= 0，那么使用缺省值；如果选择了 CONFIG_PREEMPT，缺省值为 max(10, 2*NR_CPUS)，否则缺省值为 NR_CPUS。

如果将 maxactive 设的太低也没关系；只是错过一些探测而已。在结构 kretprobe 中，注册 return probe 时 nmissed 字段为 0，每次进入被探测函数但是没有构建 return probe 所需的 kretprobe_instance 对象时会将该字段值增 1。

1.3.2 Kretprobe 入口处理函数

Kretprobes 还提供一个可选的可由用户指定的处理函数，用于在函数入口处运行。该处理函数通过设置 kretprobe 结构的 entry_handler 字段来指定。当在函数入口触发 kreprobe 设置的 kprobe 时，如果有的话会调用用户定义的 entry_handler。如果 entry_handler 返回 0（成功），那么对应的会保证在函数返回时调用一个返回处理函数。如果 entry_handler 返回一个非 0 错误，那么 Kprobes 不改动返回地址，kretprobe 也不会进一步影响对应的函数实例。

多个入口和返回处理函数的调用是配对的，通过唯一的 kretprobe_instance 对象联系在一起。此外，用户还可以指定每个返回实例的私有数据作为 kretprobe_instance 对象的一部分。可以用于在对应的用户入口和返回处理函数之间共享私有数据。每个私有数据对象的大小可以在注册 kretprobe 时设置 kretprobe 结构的 data_size 字段来指定。该数据可以通过 kretprobe_instance 对象的 data 字段来访问。

如果进入了被探测的函数，但是没有可用的 kretprobe_instance 对象，那么除了会增加 nmissed 计数外，用户指定的 entry_handler 也不会被调用。

2. 支持的体系结构

当前 kprobes、jprobes 和 return probes 支持下列体系结构：

- i386
- x86_64 (AMD-64, EM64T)
- ppc64
- ia64 (不支持对指令 slot1 的探测)
- sparc64 (尚未实现 Return probes)
- arm
- ppc

3. 配置 Kprobes

使用 make menuconfig/xconfig/oldconfig 配置内核时，将 **CONFIG_KPROBES** 设置为"y"。在菜单"**Instrumentation Support**"下查找"**Kprobes**"项。

要加载/卸载基于 Kprobes 测试系统的模块，选中"Loadable module support"(CONFIG_MODULES)和"Module unloading" (CONFIG_MODULE_UNLOAD)。

确保 **CONFIG_KALLSYMS** 甚至 **CONFIG_KALLSYMS_ALL** 设置为"y"，因为内核的 kprobe 地址解析代码会用到 kallsyms_lookup_name()。

如果要在一个函数中间插入一个 probe，**"Compile the kernel with debug info"(CONFIG_DEBUG_INFO)**会有所帮助，这样可以使用"**objdump -d -l vmlinux**"查看源码和目标代码的映射。

4. API 参考

Kprobes API 中包含每种探测类型的"register"函数和"unregister"函数。API 还包含注册/注销一组探测的"register_*probes"和"unregister_*probes"函数。下面是这些函数以及常用的相关探测处理函数的简明手册。实际应用可以参考 samples/kprobes/子目录下的例子程序。

4.1 register_kprobe

```
#include <linux/kprobes.h>
int register_kprobe(struct kprobe *kp);
```

在地址 kp->addr 处设置一个断点。当执行到断点时，Kprobe 调用 kp->pre_handler。单步执行完被探测的指令后，Kprobe 调用 kp->post_handler。如果执行 kp->pre_handler 或者 kp->post_handler 过程中或者被探测指令单步执行过程中出现故障，Kprobes 调用 kp->fault_handler。任意的处理函数都可以为 NULL。如果 kp->flags 设置了 KPROBE_FLAG_DISABLED，那么 kp 会被注册，但是被禁止，只能通过 enable_kprobe(kp)激活以后才能调用处理函数。

注意：

1. 通过 struct kprobe 中的"symbol_name"字段，内核处理探测点地址的解析。现在可以这样做：

```
                    kp.symbol_name = "symbol_name";
```

（64 位 powerpc 中复杂的结构比如函数描述符的处理是透明的）

2. 如果知道符号内部要安装断点的偏移，则使用 struct kprobe 的"offset"字段。该字段用于计算探测点

3. 指定"symbol_name"或者"addr"。如果同时指定两个，kprobe 的注册会失败并返回-EINVAL

4. 对于 CISC 体系架构（例如 i386 和 x86_64），kprobes 代码不会验证是否 kprobe.addr 位于指令边界。使用"offset"时要谨慎

register_kprobe()成功返回 0，失败返回负的 errno 值

pre-handler(**kp->pre_handler**)：

```
                    #include <linux/kprobes.h>
                    #include <linux/ptrace.h>
                    int pre_handler(struct kprobe *p, struct pt_regs *regs);
```

参数 p 指向与断点关联的 kprobe，regs 指向包含了当执行到断点时寄存器内容的结构。这里一般返回 0，除非你想用 Kprobes 做更多的事情

post-handler(**kp->post_handler**)：

```
                    #include <linux/kprobes.h>
                    #include <linux/ptrace.h>
                    void post_handler(struct kprobe *p, struct pt_regs *regs,
                            unsigned long flags);
```

p 和 regs 的用法与 pre_handler 的描述相同；flags 总是为 0

fault-handler (**kp->fault_handler**)：

```
                    #include <linux/kprobes.h>
                    #include <linux/ptrace.h>
                    int fault_handler(struct kprobe *p, struct pt_regs *regs, int trapnr);
```

p 和 regs 的用法与 pre_handler 的描述相同；trapnr 是体系结构相关的对应故障的 trap 号（例如在 i386 上，13 表示通用保护异常，14 表示页面故障）。如果成功处理异常则返回 1

4.2 register_jprobe

```
            #include <linux/kprobes.h>
            int register_jprobe(struct jprobe *jp)
```

在地址 jp->kp.addr 设置一个断点，该地址必须是函数第一条指令的地址。当执行到断点时，Kprobes 执行地址 jp->entry 指定的处理函数

处理函数的参数列表和返回类型要与被探测函数相同；处理函数返回前，必须调用 jprobe_return()。（处理函数不会真正返回，因为 jprobe_return()将控制权返回给 Kprobes。）如果被探测的函数声明为 asmlinkage 或者其它影响参数传递的声明，处理函数的声明也必须匹配

register_jprobe()成功返回 0，失败返回负的 errno 值

4.3 register_kretprobe

```
#include <linux/kprobes.h>
int register_kretprobe(struct kretprobe *rp);
```

建立 return probe 以探测地址 rp->kp.addr 指定的函数。当函数返回时，Kprobes 调用 rp->handler。必须在调用 register_kretprobe()之前将 rp->maxactive 设置为适当的值；参考“Return Probe 如何工作”获取更多信息

register_kretprobe()成功返回 0，失败返回负的 errno 值

return-probe 处理函数(**rp->handler**)：

```
#include <linux/kprobes.h>
#include <linux/ptrace.h>
int kretprobe_handler(struct kretprobe_instance *ri, struct pt_regs *regs);
```

regs 参数的用法与 kprobe.pre_handler 中的相同；ri 指向 kretprobe_instance 对象，该对象中有几个重要字段：

```
- ret_addr    ：返回地址
- rp          ：指向对应的 kretprobe 对象
- task        ：指向对应的 task 结构
- data        ：指向 return-instance 私有数据；参考"Kretprobe entry-handler"获
                取更多信息
```

宏 regs_return_value(regs)提供了从合适的注册器获取返回地址的功能，由体系机构相关的 ABI 定义

当前忽略处理函数的返回值

4.4 unregister_*probe

```
#include <linux/kprobes.h>
void unregister_kprobe(struct kprobe *kp);
void unregister_jprobe(struct jprobe *jp);
void unregister_kretprobe(struct kretprobe *rp);
```

删除指定的探测。注销函数可以在注册探测后的任意时刻调用。

注意：如果函数发现错误的探测（例如，一个未注册的探测），会清除探测的 addr 字段。

4.5 register_*probes

```
#include <linux/kprobes.h>
int register_kprobes(struct kprobe **kps, int num);
int register_kretprobes(struct kretprobe **rps, int num);
int register_jprobes(struct jprobe **jps, int num);
```

注册指定数组中的 num 个 probe。如果注册过程中发生错误，数组中所有的 probe，从最开始到出现错误的 probe，在 register_*probes 返回之前被安全地注销

```
- kps/rps/jps  ：指向*probe 数据结构数组的指针
```

```
            - num            ：数组中的元素数量
```

注意：必须分配（或定义）指针数组，并在使用这些函数之前设置好数组元素

4.6 unregister_*probes

```
            #include <linux/kprobes.h>
            void unregister_kprobes(struct kprobe **kps, int num);
            void unregister_kretprobes(struct kretprobe **rps, int num);
            void unregister_jprobes(struct jprobe **jps, int num);
```

立即删除指定数组中的 num 个 probe

注意：如果函数在指定的数组中检测到错误的 probes（例如，未注册的 probe），会将错误 probe 的 addr 字段清除；其他的 probe 会安全地注销掉

4.7 disable_*probe

```
            #include <linux/kprobes.h>
            int disable_kprobe(struct kprobe *kp);
            int disable_kretprobe(struct kretprobe *rp);
            int disable_jprobe(struct jprobe *jp);
```

临时禁止指定的*probe。可以通过 enable_*probe()激活禁止的 probe。必须指定注册了的 probe

4.8 enable_*probe

```
            #include <linux/kprobes.h>
            int enable_kprobe(struct kprobe *kp);
            int enable_kretprobe(struct kretprobe *rp);
            int enable_jprobe(struct jprobe *jp);
```

激活由 disable_*probe()禁止的*probe。指定的 probe 必须已近注册

5. Kprobes 的特点和不足

Kprobes 支持在同一地址上的多个探测。但是当前同一时间不能对同一函数有多个 jprobe。

通常，可以在内核中的任何位置安装 probe。特别地，可以探测中断处理程序。本节讨论一些特殊情况。

如果在实现 Kprobes 的代码（kernel/kprobes.c 和 arch/*/kernel/kprobes.c 文件，以及类似 do_page_fault 和 notifier_call_chain 的函数）中安装探测，函数 register_*probe 会返回-EINVAL。

如果在内联函数中安装探测，Kprobes 会尝试跟踪函数所有的内联实例，并安装探测。gcc 可能会自动内联某个函数而不需要显示指明，所以可能不能像预期那样看到 probe 被触发。

探测处理函数可以修改被探测的函数的环境——例如，修改内核数据结构，或者修改

pt_regs 结构（从断点返回时恢复寄存器）的内容。所以，Kprobes 可用于安装 bug fix 或者产生错误以便测试。当然，Kprobes 不能区分故意引发的故障和偶然出现的故障。不要沉溺于探测。

Kprobes 会阻止探测处理函数叠加——例如，探测 printk()，然后从探测处理函数中调用 printk()。如果处理函数执行探测，第二个探测处理函数不会在这个实例中运行，而是增加第二个探测中的 kprobe.nmissed 字段。

Linux v2.6.15-rc1 支持在不同 CPU 上并发运行多个处理函数（或者同一个处理函数的多个实例）。

Kprobes 只有在注册或者注销时才使用互斥量或者分配内存。

探测处理函数运行时抢占被禁止。在不同的体系结构上，处理函数运行时可能还会禁止中断。在任何一种情况下，处理程序都不能释放 CPU（例如，尝试获取一个信号量）。

因为 return probe 的实现是将返回地址替换为 trampoline 地址，栈的反向跟踪和调用 __builtin_return_address()通常会产生 trampoline 地址来替代 kretprobed 函数的实际的返回地址。（据我们所知，__builtin_return_address()只用于测试和错误报告。）

如果函数调用的次数与返回次数不匹配，那么在函数上注册 return probe 的结果可能不符合预期。这时会得到：

```
kretprobe BUG!: Processing kretprobe d000000000041aa8 @ c00000000004f48c
```

利用这条信息可以校正引发问题的 kretprobe 实例。可以处理 do_exit()的情况；do_execve()和 do_fork()没有问题。当前还不知道是否有其它情况会产生该问题。

如果进入或者退出一个函数时，CPU 正运行在栈上，而该栈不属于当前进程，那么在该函数上注册一个 return probe 的结果可能不符合预期。因此，Kprobes 不支持在 x86_64 版本的__switch_to()上安装 return probes（或者 kprobes、jprobes）；注册函数返回 return -EINVAL。

6. Probe 的开销

在 2005 年使用的一个典型的 CPU 上，一次 kprobe 触发需要 0.5 到 1.0 微秒的处理时间。特别地，一个重复触发同一个探测点的 benchmark，每次触发一个简单的处理函数，每秒会报告 1 到 2 百万次触发，这个数字根据不同的体系结构会有所不同。触发一个 jprobe 或者 return-probe 通常需要比触发一个 kprobe 多出 50-75%的时间。如果则一个函数上设置一个 return probe，那么向函数的入口添加一个 kprobe 基本上不会增加开销。

下面是不同体系结构上的开销统计样例（以 usec 为单位）。k 表示 kprobe，j 表示 jprobe，r 表示 return probe；kr 表示同一个函数上的 kprobe + return probe，jr 表示同一个函数上的 jprobe + return probe：

```
i386: Intel Pentium M, 1495 MHz, 2957.31 bogomips
        k = 0.57 usec; j = 1.00; r = 0.92; kr = 0.99; jr = 1.40

x86_64: AMD Opteron 246, 1994 MHz, 3971.48 bogomips
        k = 0.49 usec; j = 0.76; r = 0.80; kr = 0.82; jr = 1.07

ppc64: POWER5 (gr), 1656 MHz (SMT disabled, 1 virtual CPU per physical CPU)
        k = 0.77 usec; j = 1.31; r = 1.26; kr = 1.45; jr = 1.99
```

7. TODO

a. SystemTap (http://sourceware.org/systemtap)：提供了基于探测的测试系统的简化的编程接口。尝试一下

b. sparc64 内核的 Return Probes

c. 支持其它体系结构

d. 用户空间的探测

e. 观察点上的探测（在引用数据时触发）

8. Kprobes 示例

参考 samples/kprobes/kprobe_example.c

9. Jprobes 示例

参考 samples>kprobes>jprobe_example

10. Kretprobes 示例

参考 samples>kprobes>kretprobe_example

关于 Kprobes 的更多信息，参考下列网址：

http://www-106.ibm.com/developerworks/library/l-kprobes.html?ca=dgr-lnxw42Kprobe

http://www.redhat.com/magazine/005mar05/features/kprobes/

http://www-users.cs.umn.edu/~boutcher/kprobes/

http://www.linuxsymposium.org/2006/linuxsymposium_procv2.pdf (pages 101-115)

附录：kprobes 的 debugfs 接口

在最近版本的内核中(> 2.6.20)，可以在**/sys/kernel/debug/kprobes/**目录（假定 debugfs 挂载在/sys/kernel/debug）下查看注册的 kprobes 的列表。

/sys/kernel/debug/kprobes/list：列出系统中所有注册的 probes

```
c015d71a  k  vfs_read+0x0
c011a316  j  do_fork+0x0
c03dedc5  r  tcp_v4_rcv+0x0
```

第 1 列给出了 probe 插入的内核地址；第 2 列给出探测的类型（**k - kprobe**、**r - kretprobe**、**j – jprobe**）；第 3 列给出了探测的 symbol+offset。如果被探测的函数在一个模块内，还会给出模块的名字。接下来给出探测的状态。如果探测的是不再正确的虚拟地址（模块初始化区域、对应于已经卸载的模块的虚拟地址），这种 probe 标记为**[GONE]**；如果探测被临时禁止，这种探测标记为**[DISABLED]**

/sys/kernel/debug/kprobes/enabled：强制打开/关闭 kprobes

提供全局开关，以强制性打开/关闭已注册的 kprobes。缺省会打开所有的 kprobes。向该文件写入"0"，会解除所有注册的 probe，直到向该文件写入"1"。注意该开关指示解除/安装所有的 kprobes，并不改变每个 probe 的禁止状态，即禁止的 kprobes（标记为[DISABLED]）并不会因为通过该开关打开所有的 kprobes 而被激活

【附录A】linux>Documentation>kprobes.txt

```
Title  : Kernel Probes (Kprobes)
Authors: Jim Keniston <jkenisto@us.ibm.com>
       : Prasanna S Panchamukhi <prasanna@in.ibm.com>

CONTENTS

1. Concepts: Kprobes, Jprobes, Return Probes
2. Architectures Supported
3. Configuring Kprobes
4. API Reference
5. Kprobes Features and Limitations
6. Probe Overhead
7. TODO
8. Kprobes Example
9. Jprobes Example
10. Kretprobes Example
Appendix A: The kprobes debugfs interface

1. Concepts: Kprobes, Jprobes, Return Probes

Kprobes enables you to dynamically break into any kernel routine and
collect debugging and performance information non-disruptively. You
can trap at almost any kernel code address, specifying a handler
routine to be invoked when the breakpoint is hit.

There are currently three types of probes: kprobes, jprobes, and
kretprobes (also called return probes).  A kprobe can be inserted
on virtually any instruction in the kernel.  A jprobe is inserted at
the entry to a kernel function, and provides convenient access to the
function's arguments.  A return probe fires when a specified function
returns.

In the typical case, Kprobes-based instrumentation is packaged as
a kernel module.  The module's init function installs ("registers")
one or more probes, and the exit function unregisters them.  A
registration function such as register_kprobe() specifies where
the probe is to be inserted and what handler is to be called when
the probe is hit.

There are also register_/unregister_*probes() functions for batch
registration/unregistration of a group of *probes. These functions
can speed up unregistration process when you have to unregister
a lot of probes at once.

The next three subsections explain how the different types of
probes work.  They explain certain things that you'll need to
know in order to make the best use of Kprobes -- e.g., the
difference between a pre_handler and a post_handler, and how
to use the maxactive and nmissed fields of a kretprobe.  But
if you're in a hurry to start using Kprobes, you can skip ahead
to section 2.
```

1.1 How Does a Kprobe Work?

When a kprobe is registered, Kprobes makes a copy of the probed instruction and replaces the first byte(s) of the probed instruction with a breakpoint instruction (e.g., int3 on i386 and x86_64).

When a CPU hits the breakpoint instruction, a trap occurs, the CPU's registers are saved, and control passes to Kprobes via the notifier_call_chain mechanism. Kprobes executes the "pre_handler" associated with the kprobe, passing the handler the addresses of the kprobe struct and the saved registers.

Next, Kprobes single-steps its copy of the probed instruction. (It would be simpler to single-step the actual instruction in place, but then Kprobes would have to temporarily remove the breakpoint instruction. This would open a small time window when another CPU could sail right past the probepoint.)

After the instruction is single-stepped, Kprobes executes the "post_handler," if any, that is associated with the kprobe. Execution then continues with the instruction following the probepoint.

1.2 How Does a Jprobe Work?

A jprobe is implemented using a kprobe that is placed on a function's entry point. It employs a simple mirroring principle to allow seamless access to the probed function's arguments. The jprobe handler routine should have the same signature (arg list and return type) as the function being probed, and must always end by calling the Kprobes function jprobe_return().

Here's how it works. When the probe is hit, Kprobes makes a copy of the saved registers and a generous portion of the stack (see below). Kprobes then points the saved instruction pointer at the jprobe's handler routine, and returns from the trap. As a result, control passes to the handler, which is presented with the same register and stack contents as the probed function. When it is done, the handler calls jprobe_return(), which traps again to restore the original stack contents and processor state and switch to the probed function.

By convention, the callee owns its arguments, so gcc may produce code that unexpectedly modifies that portion of the stack. This is why Kprobes saves a copy of the stack and restores it after the jprobe handler has run. Up to MAX_STACK_SIZE bytes are copied -- e.g., 64 bytes on i386.

Note that the probed function's args may be passed on the stack or in registers. The jprobe will work in either case, so long as the handler's prototype matches that of the probed function.

1.3 Return Probes

1.3.1 How Does a Return Probe Work?

When you call register_kretprobe(), Kprobes establishes a kprobe at the entry to the function. When the probed function is called and this probe is hit, Kprobes saves a copy of the return address, and replaces the return address with the address of a "trampoline." The trampoline is an arbitrary piece of code -- typically just a nop instruction. At boot time, Kprobes registers a kprobe at the trampoline.

When the probed function executes its return instruction, control passes to the trampoline and that probe is hit. Kprobes' trampoline handler calls the user-specified return handler associated with the kretprobe, then sets the saved instruction pointer to the saved return address, and that's where execution resumes upon return from the trap.

While the probed function is executing, its return address is stored in an object of type kretprobe_instance. Before calling register_kretprobe(), the user sets the maxactive field of the kretprobe struct to specify how many instances of the specified function can be probed simultaneously. register_kretprobe() pre-allocates the indicated number of kretprobe_instance objects.

For example, if the function is non-recursive and is called with a spinlock held, maxactive = 1 should be enough. If the function is non-recursive and can never relinquish the CPU (e.g., via a semaphore or preemption), NR_CPUS should be enough. If maxactive <= 0, it is set to a default value. If CONFIG_PREEMPT is enabled, the default is max(10, 2*NR_CPUS). Otherwise, the default is NR_CPUS.

It's not a disaster if you set maxactive too low; you'll just miss some probes. In the kretprobe struct, the nmissed field is set to zero when the return probe is registered, and is incremented every time the probed function is entered but there is no kretprobe_instance object available for establishing the return probe.

1.3.2 Kretprobe entry-handler

Kretprobes also provides an optional user-specified handler which runs on function entry. This handler is specified by setting the entry_handler field of the kretprobe struct. Whenever the kprobe placed by kretprobe at the function entry is hit, the user-defined entry_handler, if any, is invoked. If the entry_handler returns 0 (success) then a corresponding return handler is guaranteed to be called upon function return. If the entry_handler returns a non-zero error then Kprobes leaves the return address as is, and the kretprobe has no further effect for that particular function instance.

Multiple entry and return handler invocations are matched using the unique kretprobe_instance object associated with them. Additionally, a user may also specify per return-instance private data to be part of each kretprobe_instance object. This is especially useful when sharing private data between corresponding user entry and return handlers. The size of each private data object can be specified at kretprobe registration time by setting the data_size field of the kretprobe struct. This data can be accessed through the data field of each kretprobe_instance object.

In case probed function is entered but there is no kretprobe_instance object available, then in addition to incrementing the nmissed count, the user entry_handler invocation is also skipped.

2. Architectures Supported

Kprobes, jprobes, and return probes are implemented on the following architectures:

- i386
- x86_64 (AMD-64, EM64T)
- ppc64
- ia64 (Does not support probes on instruction slot1.)
- sparc64 (Return probes not yet implemented.)
- arm
- ppc

3. Configuring Kprobes

When configuring the kernel using make menuconfig/xconfig/oldconfig, ensure that CONFIG_KPROBES is set to "y". Under "Instrumentation Support", look for "Kprobes".

So that you can load and unload Kprobes-based instrumentation modules, make sure "Loadable module support" (CONFIG_MODULES) and "Module unloading" (CONFIG_MODULE_UNLOAD) are set to "y".

Also make sure that CONFIG_KALLSYMS and perhaps even CONFIG_KALLSYMS_ALL are set to "y", since kallsyms_lookup_name() is used by the in-kernel kprobe address resolution code.

If you need to insert a probe in the middle of a function, you may find it useful to "Compile the kernel with debug info" (CONFIG_DEBUG_INFO), so you can use "objdump -d -l vmlinux" to see the source-to-object code mapping.

4. API Reference

The Kprobes API includes a "register" function and an "unregister" function for each type of probe. The API also includes "register_*probes" and "unregister_*probes" functions for (un)registering arrays of probes. Here are terse, mini-man-page specifications for these functions and the associated probe handlers that you'll write. See the files in the samples/kprobes/ sub-directory for examples.

4.1 register_kprobe

```
#include <linux/kprobes.h>
int register_kprobe(struct kprobe *kp);
```

Sets a breakpoint at the address kp->addr. When the breakpoint is hit, Kprobes calls kp->pre_handler. After the probed instruction is single-stepped, Kprobe calls kp->post_handler. If a fault occurs during execution of kp->pre_handler or kp->post_handler, or during single-stepping of the probed instruction, Kprobes calls kp->fault_handler. Any or all handlers can be NULL. If kp->flags is set KPROBE_FLAG_DISABLED, that kp will be registered but disabled, so, it's handlers aren't hit until calling enable_kprobe(kp).

NOTE:
1. With the introduction of the "symbol_name" field to struct kprobe, the probepoint address resolution will now be taken care of by the kernel. The following will now work:

```
	kp.symbol_name = "symbol_name";
```

(64-bit powerpc intricacies such as function descriptors are handled transparently)

2. Use the "offset" field of struct kprobe if the offset into the symbol to install a probepoint is known. This field is used to calculate the probepoint.

3. Specify either the kprobe "symbol_name" OR the "addr". If both are specified, kprobe registration will fail with -EINVAL.

4. With CISC architectures (such as i386 and x86_64), the kprobes code does not validate if the kprobe.addr is at an instruction boundary. Use "offset" with caution.

register_kprobe() returns 0 on success, or a negative errno otherwise.

User's pre-handler (kp->pre_handler):
```
#include <linux/kprobes.h>
#include <linux/ptrace.h>
int pre_handler(struct kprobe *p, struct pt_regs *regs);
```

Called with p pointing to the kprobe associated with the breakpoint, and regs pointing to the struct containing the registers saved when the breakpoint was hit. Return 0 here unless you're a Kprobes geek.

User's post-handler (kp->post_handler):
```
#include <linux/kprobes.h>
#include <linux/ptrace.h>
void post_handler(struct kprobe *p, struct pt_regs *regs,
	unsigned long flags);
```

p and regs are as described for the pre_handler. flags always seems to be zero.

User's fault-handler (kp->fault_handler):
```
#include <linux/kprobes.h>
#include <linux/ptrace.h>
int fault_handler(struct kprobe *p, struct pt_regs *regs, int trapnr);
```

p and regs are as described for the pre_handler. trapnr is the architecture-specific trap number associated with the fault (e.g., on i386, 13 for a general protection fault or 14 for a page fault). Returns 1 if it successfully handled the exception.

4.2 register_jprobe

```
#include <linux/kprobes.h>
int register_jprobe(struct jprobe *jp)
```

Sets a breakpoint at the address jp->kp.addr, which must be the address of the first instruction of a function. When the breakpoint is hit, Kprobes runs the handler whose address is jp->entry.

The handler should have the same arg list and return type as the probed function; and just before it returns, it must call jprobe_return(). (The handler never actually returns, since jprobe_return() returns control to Kprobes.) If the probed function is declared asmlinkage or anything else that affects how args are passed, the handler's declaration must match.

register_jprobe() returns 0 on success, or a negative errno otherwise.

4.3 register_kretprobe

```
#include <linux/kprobes.h>
int register_kretprobe(struct kretprobe *rp);
```

Establishes a return probe for the function whose address is rp->kp.addr. When that function returns, Kprobes calls rp->handler. You must set rp->maxactive appropriately before you call register_kretprobe(); see "How Does a Return Probe Work?" for details.

register_kretprobe() returns 0 on success, or a negative errno otherwise.

User's return-probe handler (rp->handler):
```
#include <linux/kprobes.h>
#include <linux/ptrace.h>
int kretprobe_handler(struct kretprobe_instance *ri, struct pt_regs *regs);
```

regs is as described for kprobe.pre_handler. ri points to the kretprobe_instance object, of which the following fields may be of interest:
- ret_addr: the return address
- rp: points to the corresponding kretprobe object
- task: points to the corresponding task struct
- data: points to per return-instance private data; see "Kretprobe entry-handler" for details.

The regs_return_value(regs) macro provides a simple abstraction to extract the return value from the appropriate register as defined by the architecture's ABI.

The handler's return value is currently ignored.

4.4 unregister_*probe

```
#include <linux/kprobes.h>
void unregister_kprobe(struct kprobe *kp);
void unregister_jprobe(struct jprobe *jp);
void unregister_kretprobe(struct kretprobe *rp);
```

Removes the specified probe. The unregister function can be called at any time after the probe has been registered.

NOTE:
If the functions find an incorrect probe (ex. an unregistered probe), they clear the addr field of the probe.

4.5 register_*probes

```
#include <linux/kprobes.h>
int register_kprobes(struct kprobe **kps, int num);
int register_kretprobes(struct kretprobe **rps, int num);
int register_jprobes(struct jprobe **jps, int num);
```

Registers each of the num probes in the specified array. If any error occurs during registration, all probes in the array, up to the bad probe, are safely unregistered before the register_*probes function returns.
- kps/rps/jps: an array of pointers to *probe data structures
- num: the number of the array entries.

NOTE:
You have to allocate(or define) an array of pointers and set all of the array entries before using these functions.

4.6 unregister_*probes

```
#include <linux/kprobes.h>
void unregister_kprobes(struct kprobe **kps, int num);
void unregister_kretprobes(struct kretprobe **rps, int num);
void unregister_jprobes(struct jprobe **jps, int num);
```

Removes each of the num probes in the specified array at once.

NOTE:
If the functions find some incorrect probes (ex. unregistered probes) in the specified array, they clear the addr field of those incorrect probes. However, other probes in the array are unregistered correctly.

4.7 disable_*probe

```
#include <linux/kprobes.h>
int disable_kprobe(struct kprobe *kp);
int disable_kretprobe(struct kretprobe *rp);
int disable_jprobe(struct jprobe *jp);
```

Temporarily disables the specified *probe. You can enable it again by using enable_*probe(). You must specify the probe which has been registered.

4.8 enable_*probe

```
#include <linux/kprobes.h>
int enable_kprobe(struct kprobe *kp);
int enable_kretprobe(struct kretprobe *rp);
int enable_jprobe(struct jprobe *jp);
```

Enables *probe which has been disabled by disable_*probe(). You must specify the probe which has been registered.

5. Kprobes Features and Limitations

Kprobes allows multiple probes at the same address. Currently, however, there cannot be multiple jprobes on the same function at the same time.

In general, you can install a probe anywhere in the kernel. In particular, you can probe interrupt handlers. Known exceptions are discussed in this section.

The register_*probe functions will return -EINVAL if you attempt to install a probe in the code that implements Kprobes (mostly kernel/kprobes.c and arch/*/kernel/kprobes.c, but also functions such as do_page_fault and notifier_call_chain).

If you install a probe in an inline-able function, Kprobes makes no attempt to chase down all inline instances of the function and install probes there. gcc may inline a function without being asked, so keep this in mind if you're not seeing the probe hits you expect.

A probe handler can modify the environment of the probed function -- e.g., by modifying kernel data structures, or by modifying the contents of the pt_regs struct (which are restored to the registers upon return from the breakpoint). So Kprobes can be used, for example, to install a bug fix or to inject faults for testing. Kprobes, of course, has no way to distinguish the deliberately injected faults from the accidental ones. Don't drink and probe.

Kprobes makes no attempt to prevent probe handlers from stepping on each other -- e.g., probing printk() and then calling printk() from a probe handler. If a probe handler hits a probe, that second probe's handlers won't be run in that instance, and the kprobe.nmissed member of the second probe will be incremented.

As of Linux v2.6.15-rc1, multiple handlers (or multiple instances of the same handler) may run concurrently on different CPUs.

Kprobes does not use mutexes or allocate memory except during registration and unregistration.

Probe handlers are run with preemption disabled. Depending on the architecture, handlers may also run with interrupts disabled. In any case, your handler should not yield the CPU (e.g., by attempting to acquire a semaphore).

Since a return probe is implemented by replacing the return address with the trampoline's address, stack backtraces and calls to __builtin_return_address() will typically yield the trampoline's address instead of the real return address for kretprobed functions. (As far as we can tell, __builtin_return_address() is used only for instrumentation and error reporting.)

If the number of times a function is called does not match the number of times it returns, registering a return probe on that function may produce undesirable results. In such a case, a line:
kretprobe BUG!: Processing kretprobe d000000000041aa8 @ c00000000004f48c
gets printed. With this information, one will be able to correlate the exact instance of the kretprobe that caused the problem. We have the do_exit() case covered. do_execve() and do_fork() are not an issue. We're unaware of other specific cases where this could be a problem.

If, upon entry to or exit from a function, the CPU is running on a stack other than that of the current task, registering a return probe on that function may produce undesirable results. For this reason, Kprobes doesn't support return probes (or kprobes or jprobes) on the x86_64 version of __switch_to(); the registration functions return -EINVAL.

6. Probe Overhead

On a typical CPU in use in 2005, a kprobe hit takes 0.5 to 1.0 microseconds to process. Specifically, a benchmark that hits the same probepoint repeatedly, firing a simple handler each time, reports 1-2 million hits per second, depending on the architecture. A jprobe or return-probe hit typically takes 50-75% longer than a kprobe hit. When you have a return probe set on a function, adding a kprobe at the entry to that function adds essentially no overhead.

Here are sample overhead figures (in usec) for different architectures.
k = kprobe; j = jprobe; r = return probe; kr = kprobe + return probe on same function; jr = jprobe + return probe on same function

i386: Intel Pentium M, 1495 MHz, 2957.31 bogomips
k = 0.57 usec; j = 1.00; r = 0.92; kr = 0.99; jr = 1.40

x86_64: AMD Opteron 246, 1994 MHz, 3971.48 bogomips
k = 0.49 usec; j = 0.76; r = 0.80; kr = 0.82; jr = 1.07

ppc64: POWER5 (gr), 1656 MHz (SMT disabled, 1 virtual CPU per physical CPU)
k = 0.77 usec; j = 1.31; r = 1.26; kr = 1.45; jr = 1.99

7. TODO

a. SystemTap (http://sourceware.org/systemtap): Provides a simplified programming interface for probe-based instrumentation. Try it out.
b. Kernel return probes for sparc64.
c. Support for other architectures.
d. User-space probes.
e. Watchpoint probes (which fire on data references).

8. Kprobes Example

See samples/kprobes/kprobe_example.c

9. Jprobes Example

See samples/kprobes/jprobe_example.c

10. Kretprobes Example

See samples/kprobes/kretprobe_example.c

For additional information on Kprobes, refer to the following URLs:
http://www-106.ibm.com/developerworks/library/l-kprobes.html?ca=dgr-lnxw42Kprobe
http://www.redhat.com/magazine/005mar05/features/kprobes/
http://www-users.cs.umn.edu/~boutcher/kprobes/
http://www.linuxsymposium.org/2006/linuxsymposium_procv2.pdf (pages 101-115)

Appendix A: The kprobes debugfs interface

With recent kernels (> 2.6.20) the list of registered kprobes is visible under the /sys/kernel/debug/kprobes/ directory (assuming debugfs is mounted at //sys/kernel/debug).

/sys/kernel/debug/kprobes/list: Lists all registered probes on the system

```
c015d71a  k  vfs_read+0x0
c011a316  j  do_fork+0x0
c03dedc5  r  tcp_v4_rcv+0x0
```

The first column provides the kernel address where the probe is inserted. The second column identifies the type of probe (k - kprobe, r - kretprobe and j - jprobe), while the third column specifies the symbol+offset of the probe. If the probed function belongs to a module, the module name is also specified. Following columns show probe status. If the probe is on a virtual address that is no longer valid (module init sections, module virtual addresses that correspond to modules that've been unloaded), such probes are marked with [GONE]. If the probe is temporarily disabled, such probes are marked with [DISABLED].

/sys/kernel/debug/kprobes/enabled: Turn kprobes ON/OFF forcibly.

Provides a knob to globally and forcibly turn registered kprobes ON or OFF. By default, all kprobes are enabled. By echoing "0" to this file, all registered probes will be disarmed, till such time a "1" is echoed to this file. Note that this knob just disarms and arms all kprobes and doesn't change each probe's disabling state. This means that disabled kprobes (marked [DISABLED]) will be not enabled if you turn ON all kprobes by this knob.

【附录 B】linux>samples>kprobes

【附录 B.1】linux>samples>kprobes>kprobe_example.c

```
/*
 * NOTE: This example is works on x86 and powerpc.
 * Here's a sample kernel module showing the use of kprobes to dump a
 * stack trace and selected registers when do_fork() is called.
 *
 * For more information on theory of operation of kprobes, see
 * Documentation/kprobes.txt
 *
 * You will see the trace data in /var/log/messages and on the console
 * whenever do_fork() is invoked to create a new process.
 */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>

/* For each probe you need to allocate a kprobe structure */
static struct kprobe kp = {
	.symbol_name	= "do_fork",
};

/* kprobe pre_handler: called just before the probed instruction is executed */
static int handler_pre(struct kprobe *p, struct pt_regs *regs)
{
#ifdef CONFIG_X86
	printk(KERN_INFO "pre_handler: p->addr = 0x%p, ip = %lx,"
			" flags = 0x%lx\n",
		p->addr, regs->ip, regs->flags);
#endif
#ifdef CONFIG_PPC
	printk(KERN_INFO "pre_handler: p->addr = 0x%p, nip = 0x%lx,"
			" msr = 0x%lx\n",
		p->addr, regs->nip, regs->msr);
#endif

	/* A dump_stack() here will give a stack backtrace */
	return 0;
}

/* kprobe post_handler: called after the probed instruction is executed */
static void handler_post(struct kprobe *p, struct pt_regs *regs,
				unsigned long flags)
{
#ifdef CONFIG_X86
	printk(KERN_INFO "post_handler: p->addr = 0x%p, flags = 0x%lx\n",
		p->addr, regs->flags);
#endif
#ifdef CONFIG_PPC
	printk(KERN_INFO "post_handler: p->addr = 0x%p, msr = 0x%lx\n",
		p->addr, regs->msr);
#endif
}

/*
 * fault_handler: this is called if an exception is generated for any
 * instruction within the pre- or post-handler, or when Kprobes
 * single-steps the probed instruction.
 */
static int handler_fault(struct kprobe *p, struct pt_regs *regs, int trapnr)
{
	printk(KERN_INFO "fault_handler: p->addr = 0x%p, trap #%dn",
		p->addr, trapnr);
	/* Return 0 because we don't handle the fault. */
	return 0;
}

static int __init kprobe_init(void)
{
	int ret;
	kp.pre_handler = handler_pre;
	kp.post_handler = handler_post;
	kp.fault_handler = handler_fault;

	ret = register_kprobe(&kp);
	if (ret < 0) {
		printk(KERN_INFO "register_kprobe failed, returned %d\n", ret);
		return ret;
	}
	printk(KERN_INFO "Planted kprobe at %p\n", kp.addr);
	return 0;
}

static void __exit kprobe_exit(void)
{
	unregister_kprobe(&kp);
	printk(KERN_INFO "kprobe at %p unregistered\n", kp.addr);
}

module_init(kprobe_init)
module_exit(kprobe_exit)
MODULE_LICENSE("GPL");
```

【附录 B.2】linux>samples>kprobes>jprobe_example.c

```
/*
 * Here's a sample kernel module showing the use of jprobes to dump
 * the arguments of do_fork().
 *
 * For more information on theory of operation of jprobes, see
 * Documentation/kprobes.txt
 *
 * Build and insert the kernel module as done in the kprobe example.
 * You will see the trace data in /var/log/messages and on the
 * console whenever do_fork() is invoked to create a new process.
 * (Some messages may be suppressed if syslogd is configured to
 * eliminate duplicate messages.)
 */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>

/*
 * Jumper probe for do_fork.
 * Mirror principle enables access to arguments of the probed routine
 * from the probe handler.
 */

/* Proxy routine having the same arguments as actual do_fork() routine */
static long jdo_fork(unsigned long clone_flags, unsigned long stack_start,
              struct pt_regs *regs, unsigned long stack_size,
              int __user *parent_tidptr, int __user *child_tidptr)
{
       printk(KERN_INFO "jprobe: clone_flags = 0x%lx, stack_size = 0x%lx,"
                     " regs = 0x%p\n",
               clone_flags, stack_size, regs);

       /* Always end with a call to jprobe_return(). */
       jprobe_return();
       return 0;
}

static struct jprobe my_jprobe = {
       .entry               = jdo_fork,
       .kp = {
              .symbol_name  = "do_fork",
       },
};

static int __init jprobe_init(void)
{
       int ret;

       ret = register_jprobe(&my_jprobe);
       if (ret < 0) {
              printk(KERN_INFO "register_jprobe failed, returned %d\n", ret);
              return -1;
       }
       printk(KERN_INFO "Planted jprobe at %p, handler addr %p\n",
              my_jprobe.kp.addr, my_jprobe.entry);
       return 0;
}

static void __exit jprobe_exit(void)
{
       unregister_jprobe(&my_jprobe);
```

```
        printk(KERN_INFO "jprobe at %p unregistered\n", my_jprobe.kp.addr);
}

module_init(jprobe_init)
module_exit(jprobe_exit)
MODULE_LICENSE("GPL");
```

【附录 B.3】linux>samples>kprobes>kretprobe_example.c

```
/*
 * kretprobe_example.c
 *
 * Here's a sample kernel module showing the use of return probes to
 * report the return value and total time taken for probed function
 * to run.
 *
 * usage: insmod kretprobe_example.ko func=<func_name>
 *
 * If no func_name is specified, do_fork is instrumented
 *
 * For more information on theory of operation of kretprobes, see
 * Documentation/kprobes.txt
 *
 * Build and insert the kernel module as done in the kprobe example.
 * You will see the trace data in /var/log/messages and on the console
 * whenever the probed function returns. (Some messages may be suppressed
 * if syslogd is configured to eliminate duplicate messages.)
 */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/ktime.h>
#include <linux/limits.h>

static char func_name[NAME_MAX] = "do_fork";
module_param_string(func, func_name, NAME_MAX, S_IRUGO);
MODULE_PARM_DESC(func, "Function to kretprobe; this module will report the"
                        " function's execution time");

/* per-instance private data */
struct my_data {
        ktime_t entry_stamp;
};

/* Here we use the entry_hanlder to timestamp function entry */
static int entry_handler(struct kretprobe_instance *ri, struct pt_regs *regs)
{
        struct my_data *data;

        if (!current->mm)
                return 1;       /* Skip kernel threads */

        data = (struct my_data *)ri->data;
        data->entry_stamp = ktime_get();
        return 0;
}

/*
 * Return-probe handler: Log the return value and duration. Duration may turn
 * out to be zero consistently, depending upon the granularity of time
 * accounting on the platform.
 */
static int ret_handler(struct kretprobe_instance *ri, struct pt_regs *regs)
{
        int retval = regs_return_value(regs);
        struct my_data *data = (struct my_data *)ri->data;
        s64 delta;
        ktime_t now;

        now = ktime_get();
        delta = ktime_to_ns(ktime_sub(now, data->entry_stamp));
        printk(KERN_INFO "%s returned %d and took %lld ns to execute\n",
                        func_name, retval, (long long)delta);
        return 0;
}

static struct kretprobe my_kretprobe = {
        .handler             = ret_handler,
        .entry_handler= entry_handler,
        .data_size           = sizeof(struct my_data),
        /* Probe up to 20 instances concurrently. */
        .maxactive           = 20,
};

static int __init kretprobe_init(void)
{
        int ret;

        my_kretprobe.kp.symbol_name = func_name;
        ret = register_kretprobe(&my_kretprobe);
        if (ret < 0) {
                printk(KERN_INFO "register_kretprobe failed, returned %d\n",
                                ret);
                return -1;
        }
        printk(KERN_INFO "Planted return probe at %s: %p\n",
                        my_kretprobe.kp.symbol_name, my_kretprobe.kp.addr);
        return 0;
}

static void __exit kretprobe_exit(void)
{
        unregister_kretprobe(&my_kretprobe);
        printk(KERN_INFO "kretprobe at %p unregistered\n",
                        my_kretprobe.kp.addr);

        /* nmissed > 0 suggests that maxactive was set too low. */
        printk(KERN_INFO "Missed probing %d instances of %s\n",
                my_kretprobe.nmissed, my_kretprobe.kp.symbol_name);
}

module_init(kretprobe_init)
module_exit(kretprobe_exit)
MODULE_LICENSE("GPL");
```

【附录 B.4】linux>samples>kprobes>Makefile

```
# builds the kprobes example kernel modules;
# then to use one (as root):  insmod <module_name.ko>

obj-$(CONFIG_SAMPLE_KPROBES) += kprobe_example.o jprobe_example.o
obj-$(CONFIG_SAMPLE_KRETPROBES) += kretprobe_example.o
```