第 4 章 调试技术

对于任何编写内核代码的人来说，最吸引他们注意的问题之一就是如何完成调试。由于内核是一个不与某个进程相关的功能集，其代码不能很轻松地放在调试器中执行，而且也不能跟踪。
本章介绍你可以用来监视内核代码和跟踪错误的技术。

用打印信息调试

最一般的调试技术就是监视，就是在应用内部合适的点加上 printf 调用。当你调试内核代码的时候，你可以用 printk 完成这个任务。

Printk

在前些章中，我们简单假设 printk 工作起来和 printf 很类似。现在是介绍一下它们之间不同的时候了。
其中一个不同点就是，printk 允许你根据它们的严重程度，通过附加不同的“记录级”来对消息分类，或赋予消息优先级。你可以用宏来指示记录级。例如，KERN_INFO，我们前面已经看到它被加在打印语句的前面，它就是一种可能的消息记录级。记录级宏展开为一个字串，在编译时和消息文本拼接在一起；这也就是为什么下面的例子中优先级和格式字串间没有逗号。这有两个 printk 的例子，一个是调试信息，一个是关键信息：
（代码）
在<linux/kernel.h>中定义了 8 种记录级别串。没有指定优先级的 printk 语句默认使用 DEFAULT_MESSAGE_LOGLEVEL 优先级，它是一个在 kernel/printk.c 中定义的整数。默认记录级的具体数值在 Linux 的开发期间曾变化过若干次，所以我建议你最好总是指定一个合适的记录级。
根据记录级，内核将消息打印到当前文本控制台上：如果优先级低于 console_loglevel 这个数值的话，该消息就显示在控制台上。如果系统同时运行了 klogd 和 syslogd，无论 console_loglevel 为何值，内核都将消息追加到/var/log/messages 中。
变量 console_loglevel 最初初始化为 DEFAULT_CONSOLE_LOGLEVEL，但可以通过 sys_syslog 系统调用修改。如 klogd 的手册所示，可以在启动 klogd 时指定-c 开关来修改这个变量。此外，你还可以写个程序来改变控制台记录级。你可以在 O’Reilly 站点上的源文件中找到我写的一个这种功能的程序，miscprogs/setlevel.c。新优先级是通过一个 1 到 8 之间的整数值指定的。
你也许需要在内核失效后降低记录级（见“调试系统故障”），这是因为失效处理代码会将 console_loglevel 提升到 15，之后所有的消息都会出现在控制台上。为看到你的调试信息，如果你运行的是内核 2.0.x 话，你需要提升记录级。内核 2.0 发行降低了 MINIMUM_CONSOLE_LOGLEVEL，而旧版本的 klogd 默认情况下要打印很多控制消息。如果你碰巧使用了这个旧版本的守护进程，除非你提升记录级，内核 2.0 会比你预期的打印出更少的消息。这就是为什么 hello.c 中使用了<1>标记，这样可以保证消息显示在控制台上。

从 1.3.43 一来的内核版本通过允许你向指定虚控制台发送消息，藉此提供一个灵活的记录策略。默认情况下，“控制台”是当前虚终端。也可以选择不同的虚终端接收消息，你只需向所选的虚终端调用 ioctl(TIOCLINUX)。如下程序，setconsole，可以用来选择哪个虚终端接收内核消息；它必须以超级用户身份运行。如果你对 ioctl 还不有把握，你可以跳过这至下一节，等到读完第 5 章“字符设备驱动程序的扩展操作”的“ioctl”一节后，再回到这里读这段代码。

（代码）

setconsole 使用了用于 Linux 专用功能的特殊的 ioctl 命令 TIOCLINUX。为了使用 TIOCLINUX，你要传递给它一个指向字节数组的指针。数组的第一个字节是所请求的子命令的编码，随后的字节依命令而不同。在 setconsole 中使用了子命令 11，后一个字节（存放在 bytes[1]中）标别虚拟控制台。TIOCLINUX 的完成介绍可以在内核源码 drivers/char/tty_io.c 中找到。

消息是如何记录的

printk 函数将消息写到一个长度为 LOG_BUF_LEN 个字节的循环缓冲区中。然后唤醒任何等待消息的进程，即那些在调用 syslog 系统调用或读取/proc/kmesg 过程中睡眠的进程。这两个访问记录引擎的接口是等价的。不过/proc/kmesg 文件更象一个 FIFO 文件，从中读取数据更容易些。一跳简单的 cat 命令就可以读取消息。

如果循环缓冲区填满了，printk 就绕到缓冲区的开始处填写新数据，覆盖旧数据。于是记录进程就丢失了最旧的数据。这个问题与利用循环缓冲区所获得的好处相比可以忽略不计。例如，循环缓冲区可以使系统在没有记录进程的情况下照样运行，同时又不浪费内存。Linux 处理消息的方法的另一个特点是，可以在任何地方调用 printk，甚至在中断处理函数里也可以调用，而且对数据量的大小没有限制。这个方法的唯一缺点就是可能丢失某些数据。

如果 klogd 正在运行，它读取内核消息并将它们分派到 syslogd，它随后检查/etc/syslog.conf 找到处理这些数据的方式。syslogd 根据一个“设施”和“优先级”切分消息；可以使用的值定义在<sys/syslog.h>中。内核消息根据相应 printk 中指定的优先级记录到 LOG_KERN 设施中。如果 klogd 没有运行，数据将保存在循环缓冲区中直到有进程来读取数据或数据溢出。

如果你不希望因监视你的驱动程序的消息而把你的系统记录搞乱，你给klogd指定-f（文件）选项或修改/etc/syslog.conf将记录写到另一个文件中。另一种方法是一种强硬方法：杀掉klogd，将消息打印到不用的虚终端上*，或者在一个不用的xterm上执行cat /proc/kmesg显示消息。

使用预处理方便监视处理

在驱动程序开发早期，printk 可以对调试和测试新代码都非常有帮助。然而当你正式发行驱动程序时，你应该去掉，或者至少关闭，这些打印语句。很不幸，你可能很快就发现，随着你想不再需要那些消息并去掉它们时，你可能又要加新功能，你又需要这些消息了。解决这些问题有几种方法——如何从全局打开和关闭消息以及如何打开和关闭个别消息。

下面给出了我处理消息所用的大部分代码，它有如下一些功能：

- 可以通过在宏名字加一个字母或去掉一个字母打开或关闭每一条语句。

* 例如，使用命令setlevel 8；set console 10 设置终端 10 显示消息。

- 通过在编译前修改 CFLAGS 变量，可以一次关闭所有消息。
- 同样的打印语句既可以用在内核态（驱动程序）也可以用在用户态（演示或测试程序）。

下面这些直接来自 scull.h 的代码片断实现了这些功能。
（代码）
符合 PDEBUG 和 PDEBUGG 依赖于是否定义了 SCULL_DEBUG，它们都和 printf 调用很类似。
为了进一步方便这个过程，在你的 Makefile 加上如下几行。
（代码）
本节所给出的代码依赖于 gcc 对 ANSI C 预编译器的扩展，gcc 可以支持带可变数目参数的宏。这种对 gcc 的依赖并不是什么问题，因为内核对 gcc 特性的依赖更强。此外，Makefile 依赖于 GNU 的 gmake；基于同样的道理，这也不是什么问题。
如果你很熟悉 C 预编译器，你可以将上面的定义扩展为可以支持“调试级”概念的，可以为每级赋一个整数（或位图），说明这一级打印多么琐碎的消息。
但是每一个驱动程序都有它自己的功能和监视需求。好的编程技巧会在灵活性和高效之间找到一个权衡点，这个我就不能说哪个对你最好了。记住，预编译器条件（还有代码中的常量表达式）只到编译时运行，你必须重新编译程序来打开或关闭消息。另一种方法就是使用 C 条件语句，它在运行时运行，因此可以让你在程序执行期间打开或关闭消息。这个功能很好，但每次代码执行系统都要进行额外的处理，甚至在消息关闭后仍然会影响性能。有时这种性能损失是无法接受的。
个人观点，尽管上面给出的宏迫使你每次要增加或去掉消息时都要重新编译，重新加载模块，但我觉得用这些宏已经很好了。

通过查询调试

上一节谈到了 printk 是如何工作的以及如何使用它。但没有谈及它的缺点。
由于 syslogd 会一直保持刷新它的输出文件，每打印一行都会引起一次磁盘操作，因此过量使用 printk 会严重降低系统性能。至少从 syslogd 的角度看是这样的。它会将所有的数据都一股脑地写到磁盘上，以防在打印消息后系统崩溃；然而，你不想因为调试信息的缘故而降低系统性能。这个问题可以通过在/etc/syslogd.conf 中记录文件的名字前加一个波折号解决，但有时你不想修改你的配置文件。如果不这样，你还可以运行一个非 klogd 的程序（如前面介绍的 cat /proc/kmesg），但这样并不能为正常操作提供一个合适的环境。
与这相比，最好的方法就是在你需要信息的时候，通过查询系统获得相关信息，而不是持续不断地产生数据。事实上，每一个 Unix 系统都提供了很多工具用来获得系统信息：ps，netstat，vmstat 等等。
有许多技术适合与驱动程序开发人员查询系统，简而言之就是，在/proc 下创建文件和使用 ioctl 驱动程序方法。

使用/proc 文件系统

Linux 中的/proc 文件系统与任何设备都没有关系－－/proc 中的文件都在被读取时有核心创建的。这些文件都是普通的文本文件，它们基本上可由普通人理解，也可被工具程序理解。例如，对于大多数 Linux 的 ps 实现而言，它都通过读取/proc 文件系统获得进程表信息的。/proc 虚拟文件的创意已由若干现代操作系统使用，且非常成功。

/proc 的当前实现可以动态创建 i 节点，允许用户模块为方便信息检索创建如何入口点。
为了在/proc 中创建一个健全的文件节点（可以 read，write，seek 等等），你需要定义 file_operations 结构和 inode_operations 结构，后者与前者有类似的作用和尺寸。创建这样一个 i 节点比起创建整个字符设备并没有什么不同。我们这里不讨论这个问题，如果你感兴趣，你可以在源码树 fs/proc 中获得进一步细节。
与大多数/proc 文件一样，如果文件节点仅仅用来读，创建它们是比较容易的，我将这里介绍这一技术。很不幸，这一技术只能在 Linux 2.0 及其后续版本中使用。
这里是创建一个称为/proc/scullmem 文件的 scull 代码，这个文件用来获取 scull 使用的内存信息。
（代码）
填写/proc 文件非常容易。你的函数获取一个空闲页面填写数据；它将数据写进缓冲区并返回所写数据的长度。其他事情都由/proc 文件系统处理。唯一的限制就是所写的数据不能超过 PAGE_SIZE 个字节（宏 PAGE_SIZE 定义在头文件<asm/page.h>中；它是与体系结构相关的，但你至少可以它有 4KB 大小）。
如果你需要写多于一个页面的数据，你必须实现功能健全的文件。
注意，如果一个正在读你的/proc文件的进程发出了若干read调用，每一个都获取新数据，尽管只有少量数据被读取，你的驱动程序每次都要重写整个缓冲区。这些额外的工作会使系统性能下降，而且如果文件产生的数据与下一次的不同，以后的read调用要重新装配不相关的部分，这一会造成数据错位。事实上，由于每个使用C库的应用程序都大块地读取数据，性能并不是什么问题。然而，由于错位时有发生，它倒是一个值得考虑的问题。在获取数据后，库调用至少要调用 1 次read－－只有当read返回 0 时才报告文件尾。如果驱动程序碰巧比前面产生了更多的数据，系统就返回到用户空间额外的字节并且与前面的数据块是错位的。我们将在第 6 章“时间流”的“任务队列”一节中涉及/proc/jiq*，那时我们还会遇到错位问题。
cleanup_module 中应该使用下面的语句注销/proc 节点：
（代码）
传递给函数的参数是包含要撤销文件的目录名和文件的 i 节点号。由于 i 节点号是自动分配的，在编译时是无法知道的，必须从数据结构中读取。

ioctl 方法

ioctl，下一章将详细讨论，是一个系统调用，它可以操做在文件描述符上；它接收一个“命令”号和（可选的）一个参数，通常这是一个指针。
做为替代/proc 文件系统的方法，你可以为调试实现若干 ioctl 命令。这些命令从驱动程序空间复制相关数据到进程空间，在进程空间里检查这些数据。
只有使用 ioctl 获取信息比起/proc 来要困难一些，因为你一个程序调用 ioctl 并显示结果。必须编写这样的程序，还要编译，保持与你测试的模块间的一致性等。
不过有时候这是最好的获取信息的方法，因为它比起读/proc 来要快得多。如果在数据写到屏幕前必须完成某些处理工作，以二进制获取数据要比读取文本文件有效得多。此外，ioctl 不限制返回数据的大小。
ioctl 方法的一个优点是，当调试关闭后调试命令仍然可以保留在驱动程序中。/proc 文件对任何查看这个目录的人都是可见的，然而与/proc 文件不同，未公开的 ioctl 命令通常都不会被注意到。此外，如果驱动程序有什么异常，它们仍然可以用来调试。唯一的缺点就是模块

*

会稍微大一些。

通过监视调试

有时你遇到的问题并不特别糟，通过在用户空间运行应用程序来查看驱动程序与系统之间的交互过程可以帮助你捕捉到一些小问题，并可以验证驱动程序确实工作正常。例如，看到 scull 的 read 实现如何处理不同数据量的 read 请求后，我对 scull 更有信心。
有许多方法监视一个用户态程序的工作情况。你可以用调试器一步步跟踪它的函数，插入打印语句，或者用 strace 运行程序。在实际目的是查看内核代码时，最后一项技术非常有用。strace 命令是一个功能非常强大的工具，它可以现实程序所调用的所有系统调用。它不仅可以显示调用，而且还能显示调用的参数，以符号方式显示返回值。当系统调用失败时，错误的符号值（如，ENOMEM）和对应的字串（Out of memory）同时显示。strace 还有许多命令行选项；最常用的是-t，它用来显示调用发生的时间，-T，显示调用所花费的时间，以及-o，将输出重定向到一个文件中。默认情况下，strace 将所有跟踪信息打印到 stderr 上。
strace 从内核接收信息。这意味着一个程序无论是否按调试方式编译（用 gcc 的-g 选项）或是被去掉了符号信息都可以被跟踪。与调试器可以连接到一个运行进程并控制它类似，你还可以跟踪一个已经运行的进程。
跟踪信息通常用来生成错误报告报告给应用开发人员，但是对内核编程人员来说也一样非常有用。我们可以看到系统调用是如何执行驱动程序代码的；strace 允许我们检查每一次调用输入输出的一致性。
例如，下面的屏幕输出给出了命令 ls /dev > /dev/scull0 的最后几行：
（代码）
很明显，在 ls 完成目标目录的检索后首次对 write 的调用中，它试图写 4KB。很奇怪，只写了 4000 个字节，接着重试这一操作。然而，我们知道 scull 的 write 实现每次只写一个量子，我在这里看到了部分写。经过若干步骤之后，所有的东西都清空了，程序正常退出。
另一个例子，让我们来读 scull 设备：
（代码）
正如所料，read 每次只能读到 4000 个字节，但是数据总量是不变的。注意本例中重试工作是如何组织的，注意它与上面写跟踪的对比。wc 专门为快速读数据进行了优化，它绕过了标准库，以便每次用一个系统调用读取更多的数据。你可以从跟踪的 read 行中看到 wc 每次要读 16KB。
Unix 专家可以在 strace 的输出中找到很多有用信息。如果你被这些符号搞得满头雾水，我可以只看文件方法（open，read 等等）是如何工作的。
个人认为，跟踪工具在查明系统调用的运行时错误过程中最有用。通常应用或演示程序中的 perror 调用不足以用来调试，而且对于查明到底是什么样的参数触发了系统调用的错误也很有帮助。

调试系统故障

即便你用了所有监视和调试技术，有时候驱动程序中依然有错误，当这样的驱动程序执行会会造成系统故障。当这种情况发生时，获取足够多的信息来解决问题是至关重要的。
注意，“故障”不意味着“panic”。Linux 代码非常鲁棒，可以很好地响应大部分错误：故障通常会导致当前进程的终止，但系统继续运行。如果在进程上下文之外发生故障，或是组成

系统的重要部件发生故障时，系统可能 panic。但问题出在驱动程序时，通常只会导致产生故障的进程终止——即那个使用驱动程序的进程。唯一不可恢复的损失就是当进程被终止时，进程上下文分配的内存丢失了；例如，由驱动程序通过 kmalloc 分配的动态链表可能丢失。然而，由于内核会对尚是打开的设备调用 close，你的驱动程序可以释放任何有 open 方法分配的资源。

我们已经说过，当内核行为异常时会在控制台上显示一些有用的信息。下一节将解释如何解码和使用这些消息。尽管它们对于初学者来说相当晦涩，处理器的给出数据都是些很有意思的信息，通常无需额外测试就可以查明程序错误。

Oops 消息

大部分错误都是 NULL 指针引用或使用其他不正确的指针数值。这些错误通常会导致一个 oops 消息。

由处理器使用的地址都是“虚”地址，而且通过一个复杂的称为页表（见第 13 章“Mmap 和 DMA”中的“页表”一节）的结构映射为物理地址。当引用一个非法指针时，页面映射机制就不能将地址映射到物理地址，并且处理器向操作系统发出一个“页面失效”。如果地址确实是非法的，内核就无法从失效地址上“换页”；如果此时处理在超级用户太，系统于是就产生一个“oops”。值得注意的是，在版本 2.1 中内核处理失效的方式有所变化，它可以处理在超级用户态的非法地址引用了。新实现将在第 17 章“最近发展”的“处理内核空间失效”中介绍。

oops 显示故障时的处理器状态，模块 CPU 寄存器内容，页描述符表的位置，以及其他似乎不能理解的信息。这些是由失效处理函数（arch/*/kernel/traps.c）中的 printk 语句产生的，而且象前面“Printk”一节介绍的那样进行分派。

让我们看看这样一个消息。这里给出的是传统个人电脑（x86 平台），运行 Linux 2.0 或更新版本的 oops——版本 1.2 的输出稍有不同。

（代码）

上面的消息是在一个有意加入错误的失效模块上运行 cat 所至。fault.c 崩溃如下代码：

（代码）

由于 read 从它的小缓冲区（faulty_buf）复制数据到用户空间，我们希望读一小块文件能够工作。然而，每次读出多于 1KB 的数据会跨越页面边界，如果访问了非法页面 read 就会失败。事实上，前面给出的 oops 是在请求一个 4KB 大小的 read 时发生的，这条消息在 /var/log/messages（syslogd 默认存放内核消息的文件）的 oops 消息前给出了：

（代码）

同样的 cat 命令却不能在 Alpha 上产生 oops，这是因为从 faulty_buf 读取 4KB 字节没有超出页边界（Alpha 上的页面大小是 8KB，缓冲区正好在页面的起始位置附近）。如果在你的系统上读取 faulty 没有产生 oops，试试 wc，或者给 dd 显式地指定块大小。

使用 ksymoops

oops 消息的最大问题就是十六进制数值对于程序员来说没什么意义；需要将它们解析为符号。

内核源码通过其所包含的 ksymoops 工具帮助开发人员——但是注意，版本 1.2 的源码中没有这个程序。该工具将 oops 消息中的数值地址解析为内核符号，但只限于 PC 机产生的 oops

消息。由于消息本身就是处理器相关的，每一体系结构都有其自身的消息格式。
ksymoops 从标准输入获得 oops 消息，并从命令行内核符号表的名字。符号表通常就是/usr/src/linux/System.map。程序以更可读的方式打印调用轨迹和程序代码，而不是最原始的 oops 消息。下面的片断就是用上一节的 oops 消息得出的结果：
（代码）
由 ksymoops 反汇编出的代码给出了失效的指令和其后的指令。很明显－－对于那些知道一点汇编的人－－repz movsl 指令（REPeat till cx is Zero, MOVe a String of Longs）用源索引（esi，是 0x202e000）访问了一个未映射页面。用来获得模块信息的 ksymoops -m 命令给出，模块映射到一个在 0x0202dxxx 的页面上，这也确认乐 esi 确实超出了范围。
由于 faulty 模块所占用的内存不在系统表中，被解码的调用轨迹还给出了两个数值地址。这些值可以手动补充，或是通过 ksyms 命令的输出，或是在/proc/ksyms 中查询模块的名字。然而对于这个失效，这两个地址并不对应与代码地址。如果你看了 arch/i386/kernel/traps.c，你就发现，调用轨迹是从整个堆栈并利用一些启发式方法区分数据值（本地变量和函数参数）和返回地址获得的。调用轨迹中只给出了引用内核代码的地址和引用模块的地址。由于模块所占页面既有代码也有数据，错综复杂的栈可能会漏掉启发式信息，这就是上面两个 0x202xxxx 地址的情况。
如果你不愿手动查看模块地址，下面这组管道可以用来创建一个既有内核又有模块符号的符号表。无论何时你加载模块，你都必须重新创建这个符号表。
（代码）
这个管道将完整的系统表与/proc/ksyms 中的公开内核符号混合在一起，后者除了内核符号外，还包括了当前内核里的模块符号。这些地址在 insmod 重定位代码后就出现在/proc/ksyms 中。由于这两个文件的格式不同，使用了 sed 和 awk 将所有的文本行转换为一种合适的格式。然后对这张表排序，去除重复部分，这样 ksymoops 就可以用了。
如果我们重新运行 ksymoops，它从新的符号表中截取出如下信息：
（代码）
正如你所见到的，当跟踪与模块有关的 oops 消息时，创建一个修订的系统表是很有助益的：现在 ksymoops 能够对指令指针解码并完成整个调用轨迹了。还要注意，显式反汇编码的格式和 objdump 所使用的格式一样。objdump 也是一个功能强大的工具；如果你需要查看失败前的指令，你调用命令 objdump –d faulty.o。
在文件的汇编列表中，字串 faulty_read+45/60 标记为失效行。有关 objdump 的更多的信息和它的命令行选项可以参见该命令的手册。
即便你构建了你自己的修订版符号表，上面提到的有关调用轨迹的问题仍然存在：虽然 0x202xxxx 指针被解码了，但仍然是假的。
学会解码 oops 消息需要一定的经验，但是确实值得一做。用来学习的时间很快就会有所回报。不过由于机器指令的 Unix 语法与 Intel 语法不同，唯一的问题在于从哪获得有关汇编语言的文档；尽管你了解 PC 汇编语言，但你的经验都是用 Intel 语法的编程获得的。在参考书目中，我给一些有所补益的书籍。

使用 oops

使用 ksymoops 有些繁琐。你需要 C++编译器编译它，你还要构建你自己的符号表来充分发挥程序的能力，你还要将原始消息和 ksymoops 输出合在一起组成可用的信息。
如果你不想找这么多麻烦，你可以使用 oops 程序。oops 在本书的 O’Reilly FTP 站点给出的源码中。它源自最初的 ksymoops 工具，现在它的作者已经不维护这个工具了。oops 是用 C

语言写成的，而且直接查看/proc/ksyms 而无需用户每次加载模块后构建新的符号表。
该程序试图解码所有的处理器寄存器并堆栈轨迹解析为符号值。它的缺点是，它要比 ksymoops 罗嗦些，但通常你所有的信息越多，你发现错误也就越快。oops 的另一个优点是，它可以解析 x86，Alpha 和 Sparc 的 oops 消息。与内核源码相同，这个程序也按 GPL 发行。
oops 产生的输出与 ksymoops 的类似，但是更完全。这里给出前一个 oops 输出的开始部分—由于在这个 oops 消息中堆栈没保存什么有用的东西，我不认为应该显示整个堆栈轨迹：
（代码）
当你调试“真正的”模块（faulty 太短了，没有什么意义）时，将寄存器和堆栈解码是非常有益的，而且如果被调试的所有模块符号都开放出来时更有帮助。在失效时，处理器寄存器一般不会指向模块的符号，只有当符号表开放给/proc/ksyms 时，你才能输出中标别它们。
我们可以用一下步骤制作一张更完整的符号表。首先，我们不应在模块中声明静态变量，否则我们就无法用 insmod 开放它们了。第二，如下面的截取自 scull 的 init_module 函数的代码所示，我们可以用#ifdef SCULL_DEBUG 或类似的宏屏蔽 register_symtab 调用。
（代码）
我们在第 2 章“编写和运行模块”的“注册符号表”一节中已经看到了类似内容，那里说，如果模块不注册符号表，所有的全局符号就都开放。尽管这一功能仅在 SCULL_DEBUG 被激活时才有效，为了避免内核中的名字空间污染，所有的全局符号有合适的前缀（参见第 2 章的“模块与应用程序”一节）。

使用 klogd

klogd 守护进程的近期版本可以在 oops 存放到记录文件前对 oops 消息解码。解码过程只由版本 1.3 或更新版本的守护进程完成，而且只有将-k /usr/src/linux/System.map 做为参数传递给守护进程时才解码。（你可以用其他符号表文件代替 System.map）
有新的 klogd 给出的 faulty 的 oops 如下所示，它写到了系统记录中：
（代码）
我想能解码的 klogd 对于调试一般的 Linux 安装的核心来说是很好的工具。由 klogd 解码的消息包括大部分 ksymoops 的功能，而且也要求用户编译额外的工具，或是，当系统出现故障时，为了给出完整的错误报告而合并两个输出。当 oops 发生在内核时，守护进程还会正确地解码指令指针。它并不反汇编代码，但这不是问题，当错误报告给出消息时，二进制数据仍然存在，可以离线反汇编代码。
守护进程的另一个功能就是，如果符号表版本与当前内核不匹配，它会拒绝解析符号。如果在系统记录中解析出了符号，你可以确信它是正确的解码。
然而，尽管它对 Linux 用户很有帮助，这个工具在调试模块时没有什么帮助。我个人没有在开放软件的电脑里使用解码选项。klogd 的问题是它不解析模块中的符号；因为守护进程在程序员加载模块前就已经运行了，即使读了/proc/ksyms 也不会有什么帮助。记录文件中存在解析后的符号会使 oops 和 ksymoops 混淆，造成进一步解析的困难。
如果你需要使用 klogd 调试你的模块，最新版本的守护进程需要加入一些新的特殊支持，我期待它的完成，只要给内核打一个小补丁就可以了。

系统挂起

尽管内核代码中的大多数错误仅会导致一个 oops 消息，有时它们困难完全将系统挂起。如

果系统挂起了，没有消息能够打印出来。例如，如果代码遇到一个死循环，内核停止了调度过程，系统不会再响应任何动作，包括魔法键 Ctrl-Alt-Del 组合。

处理系统挂起有两个选择－－一个是防范与未然，另一个就是亡羊补牢，在发生挂起后调试代码。

通过在策略点上插入 schedule 调用可以防止死循环。schedule 调用（正如你所猜想到的）调用调度器，因此允许其他进程偷取当然进程的 CPU 时间。如果进程因你的驱动程序中的错误而在内核空间循环，你可以在跟踪到这种情况后杀掉这个进程。

在驱动程序代码中插入 schedule 调用会给程序员带来新的“问题”：函数,，以及调用轨迹中的所有函数，必须是可重入的。在正常环境下，由于不同的进程可能并发地访问设备，驱动程序做为整体是可重入的，但由于 Linux 内核是不可抢占的，不必每个函数都是可重入的。但如果驱动程序函数允许调度器中断当前进程，另一个不同的进程可能会进入同一个函数。如果 schedule 调用仅在调试期间打开，如果你不允许，你可以避免两个并发进程访问驱动程序，所以并发性倒不是什么非常重要的问题。在介绍阻塞型操作时（第 5 章的“写可重入代码”）我们再详细介绍并发性问题。

如果要调试死循环，你可以利用 Linux 键盘的特殊键。默认情况下，如果和修饰键一起按了 PrScr 键（键码是 70），系统会向当前控制台打印有关机器状态的有用信息。这一功能在 x86 和 Alpha 系统都有。Linux 的 Sparc 移植也有同样的功能，但它使用了标记为“Break/Scroll Lock”的键（键码是 30）。

每一个特殊函数都有一个名字，并如下面所示都有一个按键事件与之对应。组合键之后的括号里是函数名。

Shift-PrScr（Show_Memory）

打印若干行关于内存使用的信息，尤其是有关缓冲区高速缓存的使用情况。

Control-PrScr（Show_State）

针对系统里的每一个处理器打印一行信息，同时还打印内部进程树。对当前进程进行标记。

RightAlt-PrScr（Show_Registers）

由于它可以打印按键时的处理器寄存器内容，它是系统挂起时最重要的一个键了。如果有当前内核的系统表的话，查看指令计数器以及它如何随时间变化，对了解代码在何处循环非常有帮助。

如果想将这些函数映射到不同的键上，每一个函数名都可以做为参数传递给 loadkeys。键盘映射表可以任意修改（这是“策略无关的”）。

如果 console_loglevel 足够到的话，这些函数打印的消息会出现在控制台上。如果不是你运行了一个旧 klogd 和一个新内核的话，默认记录级应该足够了。如果没有出现消息，你可以象以前说的那样提升记录级。“足够高”的具体值与你使用的内核版本有关。对于 Linux 2.0 或更新的版本来说是 5。

即便当系统挂起时，消息也会打印到控制台上，确认记录级足够高是非常重要的。消息是在产生中断时生成的，因此即便有错的进程不释放 CPU 也可以运行－－当然，除非中断被屏蔽了，不过如果发生这种情况既不太可能也非常不幸。

有时系统看起来象是挂起了，但其实不是。例如，如果键盘因某种奇怪的原因被锁住了就会发生这种情况。这种假挂起可以通过查看你为探明此种情况而运行的程序输出来判断。我有一个程序会不断地更新 LED 显示器上的时钟，我发现这个对于验证调度器尚在运行非常有用。你可以不必依赖外部设备就可以检查调度器，你可以实现一个程序让键盘 LED 闪烁，或是不断地打开关闭软盘马达，或是不断触动扬声器－－不过我个人认为，通常的蜂鸣声很烦人，应该尽量避免。看看 ioctl 命令 KDMKTONE。O’Reilly FTP 站点上的例子程序（misc-progs/heartbeat.c）中有一个是让键盘 LED 不断闪烁的。

如果键盘不接收输入了，最佳的处理手段是从网络登录在系统中，杀掉任何违例的进程，或是重新设置键盘（用 kdb_mode -a）。然而，如果你没有网络可用来恢复的话，发现系统挂起是由键盘锁死造成的一点儿用也没有。如果情况确实是这样，你应该配置一种替代输入设备，至少可以保证正常地重启系统。对于你的计算机来说，关闭系统或重启比起所谓的按“大红钮”要更方便一些，至少它可以免去长时间地 fsck 扫描磁盘。

这种替代输入设备可以是游戏杆或是鼠标。在 sunsite.edu.cn 上有一个游戏杆重启守护进程，gpm-1.10 或更新的鼠标服务器可以通过命令行选项支持类似的功能。如果键盘没有锁死，但是却误入“原始”模式，你可以看看 kdb 包中文档介绍的一些小技巧。我建议最好在问题出现以前就看看这些文档，否则就太晚了。另一种可能是配置 gpm-root 菜单，增添一个“reboot”或“reset keyboard”菜单项；gpm-root 一个响应控制鼠标事件的守护进程，它用来在屏幕上显示菜单和执行所配置的动作。

最好，你会可以按“留意安全键”（SAK），一个用于将系统恢复为可用状态的特殊键。由于不是所有的实现都能用，当前 Linux 版本的默认键盘表中没有为此键特设一项。不过你还是可以用 loadkeys 将你的键盘上的一个键映射为 SAK。你应该看看 drivers/char 目录中的 SAK 实现。代码中的注释解释了为什么这个键在 Linux 2.0 中不是总能工作，这里我就不多说了。不过，如果你运行版本 2.1.9 或是更新的版本，你就可以使用非常可靠地留意安全键了。此外，2.1.43 及后续版本内核还有一个编译选项选择是否打开“SysRq 魔法键”；我建议你看一看 drivers/char/sysrq.c 中的代码并使用这项新技术。

如果你的驱动程序真的将系统挂起了，而且你有不知道在哪插入 schedule 调用，最佳的处理方法就是加一些打印消息，并将它们打印到控制台上（通过修改 console_loglevel 变量值）。在重演挂起过程时，最好将所有的磁盘都以只读方式安装在系统上。如果磁盘是只读的或没有安装，就不会存在破坏文件系统或使其进入不一致状态的危险。至少你可以避免在复位系统后运行 fsck。另一中方法就是使用 NFS 根计算机来测试模块。在这种情况下，由于 NFS 服务器管理文件系统的一致性，而它又不会受你的驱动程序的影响，你可以避免任何的文件系统崩溃。

使用调试器

最后一种调试模块的方法就是使用调试器来一步步地跟踪代码，查看变量和机器寄存器的值。这种方法非常耗时，应该尽可能地避免。不过，某些情况下通过调试器对代码进行细粒度的分析是非常有益的。在这里，我们所说的被调试的代码运行在内核空间——除非你远程控制内核，否则不可能一步步跟踪内核，这会使很多事情变得更加困难。由于远程控制很少用到，我们最后介绍这项技术。所幸的是，在当前版本的内核中可以查看和修改变量。

在这一级上熟练地使用调试器需要精通 gdb 命令，对汇编码有一定了解，并且有能够将源码与优化后的汇编码对应起来的能力。

不幸的是，gdb 更适合与调试核心而不是模块，调试模块化的代码需要更多的技术。这更多的技术就是 kdebug 包，它利用 gdb 的“远程调试”接口控制本地内核。我将在介绍普通调试器后介绍 kdebug。

使用 gdb

gdb 在探究系统内部行为时非常有用。启动调试器时必须假想内核就是一个应用程序。除了指定内核文件名外，你还应该在命令行中提供内存镜象文件的名字。典型的 gdb 调用如下所

示：
（代码）
第一个参数是未经压缩的内核可执行文件（在你编译完内核后，这个文件在/usr/src/linux 目录中）的名字。只有 x86 体系结构有 zImage 文件（有时称为 vmlinuz），它是一种解决 Intel 处理器实模式下只有 640KB 限制的一种技巧；而无论在哪个平台上，vmlinux 都是你所编译的未经压缩的内核。
gdb 命令行的第二个参数是是内存镜象文件的名字。与其他在/proc 下的文件类似，/proc/kcore 也是在被读取时产生的。当 read 系统调用在/proc 文件系统执行时，它映射到一个用于数据生成而不是数据读取的函数上；我们已在“使用/proc 文件系统”一节中介绍了这个功能。系统用 kcore 来表示按内存镜象文件格式存储的内核“可执行文件”；由于它要表示整个内核地址空间，它是一个非常巨大的文件，对应所有的物理内存。利用 gdb，你可以通过标准 gdb 命令查看内核标量。例如，p jiffies 可以打印从系统启动到当前时刻的时钟滴答数。
当你从 gdb 打印数据时，内核还在运行，不同数据项会在不同时刻有不同的数值；然而，gdb 为了优化对内存镜象文件的访问会将已经读到的数据缓存起来。如果你再次查看 jiffies 变量，你会得到和以前相同的值。缓存变量值防止额外的磁盘操作对普通内存镜象文件来说是对的，但对“动态”内存镜象文件来说就不是很方便了。解决方法是在你想刷新 gdb 缓存的时候执行 core-file /proc/kcore 命令；调试器将使用新的内存镜象文件并废弃旧信息。但是，读新数据时你并不总是需要执行 core-file 命令；gdb 以 1KB 的尺度读取内存镜象文件，仅仅缓存它所引用的若干块。
你不能用普通 gdb 做的是修改内核数据；由于调试器需要在访问内存镜象前运行被调试程序，它是不会去修改内存镜象文件的。当调试内核镜象时，执行 run 命令会导致在执行若干指令后导致段违例。出于这个原因，/proc/kcore 都没有实现 write 方法。
如果你用调试选项（-g）编译了内核，结果产生的 vmlinux 比没有用-g 选项的更适合于 gdb。不过要注意，用-g 选项编译内核需要大量的磁盘空间－－支持网络和很少几个设备和文件系统的 2.0 内核在 PC 上需要 11KB。不过不管怎样，你都可以生成 zImage 文件并用它来其他系统：在生成可启动镜象时由于选项-g 而加入的调试信息最终都被去掉了。如果我有足够的磁盘空间，我会一致打开-g 选项的。
在非 PC 计算机上则有不同的方法。在 Alpha 上，make boot 会在生成可启动镜象前将调试信息去掉，所以你最终会获得 vmlinux 和 vmlinux.gz 两个文件。gdb 可以使用前者，但你只能用后者启动。在 Sparc 上，默认情况下内核（至少是 2.0 内核）不会被去掉调试信息，所以你需要在将其传递给 silo（Sparc 的内核加载器）前将调试信息去掉，这样才能启动。由于尺寸的问题，无论 milo（Alpha 的内核加载器）还是 silo 都不能启动未去掉调试信息的内核。
当你用-g 选项编译内核并且用 vmlinux 和/proc/kcore 一起使用调试器，gdb 可以返回很多有关内核内部结构的信息。例如，你可以使用类似于这样的命令，p *module_list，p *module_list->next 和 p *chrdevs[4]->fops 等显示这些结构的内容。如果你手头有内核映射表和源码的话，这些探测命令是非常有用的。
另一个 gdb 可以在当前内核上执行的有用任务是，通过 disassemble 命令（它可以缩写）或是“检查指令”（x/i）命令反汇编函数。disassemble 命令的参数可以是函数名或是内存区范围，而 x/i 则使用一个内存地址做为参数，也可以用符号名。例如，你可以用 x/20i 反汇编 20 条指令。注意，你不能反汇编一个模块的函数，这是因为调试器处理 vmlinux，它并不知道你的模块的信息。如果你试图用模块的地址反汇编代码，gdb 很有可能会报告“不能访问 xxxx 处的内存（Cannot access memory at xxxx）”。基于同样的原因，你不查看属于模块的数据项。如果你知道你的变量的地址，你可以从/dev/mem 中读出它的值，但很难弄明白从系

统内存中分解出的数据是什么含义。
如果你需要反汇编模块函数，你最好对用 objdump 工具处理你的模块文件。很不幸，该工具只能对磁盘上的文件进行处理，而不能对运行中的模块进行处理；因此，objdump 中给出的地址都是未经重定位的地址，与模块的运行环境无关。
如你所见，当你的目的是查看内核的运行情况时，gdb 是一个非常有用的工具，但它缺少某些功能，最重要的一些功能就是修改内核项和访问模块的功能。这些空白将由 kdebug 包填补。

使用 kdebug

你可用从一般的 FTP 站点下的 pcmcia/extras 目录下拿到 kdebug，但是如果你想确保拿到的是最新的版本，你最好到 ftp://hyper.stanford.edu/pub/pcmcia/extras/去找。该工具与 pcmcia 没有什么关系，但是这两个包是同一个作者写的。
kdebug 是一个使用 gdb“远程调试”接口与内核通信的小工具。使用时首先向内核加载一个模块，调试器通过/dev/kdebug 访问内核数据。gdb 将该设备当成一个与被调试“应用”通信的串口设备，但它仅仅是一个用于访问内核空间的通信通道。由于模块本身运行在内核空间，它可以看到普通调试器无法访问的内核空间地址。正如你所猜想到的，模块是一个字符设备驱动程序，并且使用了主设备号动态分配技术。
kdebug 的优点在于，你无需打补丁或重新编译：无论是内核还是调试器都无需修改。你所需要做的就是编译和安装软件包，然后调用 kgdb，kgdb 是一个完成某些配置并调用 gdb，通过新接口访问内核部件结构的脚本程序。
但是，即便是 kdebug 也没有提供单步跟踪内核代码和设置断点的功能。这几乎是不可避免的，因为内核必须保持运行状态以保证系统的出于运行状态，跟踪内核代码的唯一方法就是后面将要谈到的从另外一台计算机上通过串口控制系统。不过 kgdb 的实现允许用户修改被调试应用（即当前内核）的数据项，可以传递给内核任意数目的参数，并以读写方式访问模块所属的内存区。
最后一个功能就是通过 gdb 命令将模块符号表增加到调试器内部的符号表中。这个工作是由 kgdb 完成的。然后当用户请求访问某个符号时，gdb 就知道它的地址是哪了。最终的访问是由模块里的内核代码完成的。不过要注意，kdebug 的当前版本（1.6）在映射模块化代码地址方面还有些问题。你最好通过打印一些符号并与/proc/ksyms 中的值进行比较来做些检查。如果地址没有匹配，你可以使用数值，但必须将它们强行转换为正确的类型。下面就是一个强制类型转换的例子：
（代码）
kdebug 的另一个强于 gdb 的优点是，它允许你在数据结构被修改后读取到最新的值，而不必刷新调试器的缓存；gdb 命令 set remotecache 0 可以用来关闭数据缓存。
由于 kdebug 与 gdb 使用起来很相似，这里我就不过多地罗列使用这个工具的例子了。对于知道如何使用调试器的人来说，这种例子很简单，但对于那些对调试器一无所知的人来说就很晦涩了。能够熟练地使用调试器需要时间和经验，我不准备在这里承担老师的责任。
总而言之，kdebug是一个非常好的程序。在线修改数据结构对于开发人员来说是一个非常大的进步（而且一种将系统挂起的最简单方法）。现在有许多工具可以使你的开发工作更轻松－－例如，在开发scull期间，当模块的使用计数器增长后*，我可以使用kdebug来将其复位为 0。这就不必每次都麻烦我重启机器，登录，再次启动我的应用程序等等。

* 使用计数器在模块地址空间的最开始的部分，不过这是未公开的，以后可能会发生变化。

远程调试

调试内核镜象的最后一个方法是使用 gdb 的远程调试能力。
当执行远程调试的时候，你需要两台计算机：一台运行 gdb；另一台运行你要调试的内核。这两台计算机间用普通串口连接起来。如你所料，控制 gdb 必须能够理解它所控制的内核的二进制格式。如果这两台计算机是不同的体系结构，必须将调试器编译为可以支持目标平台的。
在 2.0 中，Linux 内核的 Intel 版本不支持远程调试，但是 Alpha 和 Sparc 版本都支持。在 Alpha 版本中，你必须在编译时包含对远程调试的支持，并在启动时通过传递给内核命令行参数 kgdb=1 或只有 kgdb 打开这个功能。在 Sparc 上，始终包含了对远程调试的支持。启动选项 kgdb=ttyx 可以用来选择在哪个串口上控制内核，x 可以是 a 或 b。如果没有使用 kgdb=选项，内核就按正常方式启动。
如果在内核中打开了远程调试功能，系统在启动时就会调用一个特殊的初始化函数，配置被调试内核处理它自己的断点，并且跳转到一个编译自程序中的断点。这会暂停内核的正常执行，并将控制转移给断点服务例程。这一处理函数在串口线上等待来自于 gdb 的命令，当它获得 gdb 的命令后，就执行相应的功能。通过这一配置，程序员可以单步跟踪内核代码，设置断点，并且完成 gdb 所允许的其他任务。
在控制端，需要一个目标镜象的副本（我们假设它是 linux.img），还需要一个你要调试的模块副本。如下命令必须传递给 gdb：
file linux.img
file 命令告诉 gdb 哪个二进制文件需要调试。另一种方法是在命令行中传递镜象文件名。这个文件本身必须和运行在另一端的内核一模一样。
target remote /dev/ttyS1
这条命令通知 gdb 使用远程计算机做为调试过程的目标。/dev/ttyS1 是用来通信的本地串口，你可以指定任一设备。例如，前面介绍的 kdebug 软件包中的 kgdb 脚本使用 target remote /dev/kdebug。
add-symbol-file module.o address
如果你要调试已经加载到被控内核的模块的话，在控制系统上你需要一个模块目标文件的副本。add-symbol-file 通知 gdb 处理模块文件，假定模块代码被定位在地址 address 上了。
尽管远程调试可以用于调试模块，但你还是要加载模块，并且在模块上插入断点前还需要触发另一个断点，调试模块还是需要很多技巧的。我个人不会使用远程调试去跟踪模块，除非异步运行的代码，如中断处理函数，出了问题。