第四章 调试技术

内核编程带来了其独特的调试挑战。内核代码不能轻易地在调试器下执行，也不能轻易地被追踪，因为它是一组与特定进程无关的功能。内核代码错误也可能极其难以复现，并且可能会导致整个系统崩溃，从而破坏了用于追踪它们的大部分证据。 本章将介绍你可以在这种艰难情况下用来监控内核代码和追踪错误的技术。

4.1. 内核中的调试支持

在第二章中，我们建议你构建并安装自己的内核，而不是运行你的发行版附带的标准内核。运行自己的内核的最强大的理由之一是，内核开发者已经在内核本身中构建了几个调试功能。这些功能可以产生额外的输出并降低性能，因此它们通常不会在发行商的生产内核中启用。然而，作为一个内核开发者，你有不同的优先级，并会愉快地接受额外的内核调试支持的（最小的）开销。

在这里，我们列出了用于开发的内核应启用的配置选项。除非另有说明，所有这些选项都可以在你喜欢的任何内核配置工具的“内核黑客”菜单下找到。请注意，不是所有的架构都支持这些选项。

`CONFIG_DEBUG_KERNEL`

- 这个选项只是使其他调试选项可用；它应该被打开，但本身并不启用任何功能。

`CONFIG_DEBUG_SLAB`

- 这个关键选项在内核内存分配函数中启用了几种类型的检查；启用这些检查后，可以检测到一些内存溢出和缺少初始化错误。每个分配的内存字节在交给调用者之前被设置为0xa5，然后在释放时被设置为0x6b。如果你在驱动程序的输出（或者经常在oops列表中）看到这些“毒药”模式重复，你就会知道要寻找什么样的错误。当启用调试时，内核还在每个分配的内存对象前后放置特殊的保护值；如果这些值被改变，内核就知道有人溢出了一个内存分配，并且它会大声抱怨。还启用了对更难以察觉的错误的各种检查。

`CONFIG_DEBUG_PAGEALLOC`

- 释放时，完整的页面会从内核地址空间中移除。这个选项可能会显著降低速度，但它也可以快速指出某些类型的内存损坏错误。

`CONFIG_DEBUG_SPINLOCK`

- 启用此选项后，内核会捕获对未初始化的自旋锁的操作和各种其他错误（如解锁两次锁）。

`CONFIG_DEBUG_SPINLOCK_SLEEP`

- 此选项启用了对尝试在持有自旋锁时睡眠的检查。实际上，如果你调用了可能会睡眠的函数，即使问题的调用不会睡眠，它也会抱怨。

`CONFIG_INIT_DEBUG`

- 标记为 `__init` （或 `__initdata`）的项目在系统初始化或模块加载时间后被丢弃。此选项启用了对尝试在初始化完成后访问初始化时内存的代码的检查。

`CONFIG_DEBUG_INFO`

- 此选项导致内核被构建时包含完整的调试信息。如果你想用gdb调试内核，你需要这些信息。如果你打算使用gdb，你可能还想启用 `CONFIG_FRAME_POINTER`。

`CONFIG_MAGIC_SYSRQ`

- 启用“魔术SysRq”键。我们在本章后面的“系统挂起”部分中查看这个键。

`CONFIG_DEBUG_STACKOVERFLOW`
`CONFIG_DEBUG_STACK_USAGE`

- 这些选项可以帮助追踪内核栈溢出。栈溢出的明确标志是一个没有任何合理回溯的oops列表。第一个选项向内核添加了显式的溢出检查；第二个选项使内核监视栈使用情况，并通过魔术SysRq键提供一些统计信息。

`CONFIG_KALLSYMS`

- 此选项（在“通用设置/标准功能”下）导致内核符号信息被构建到内核中；它默认是启用的。符号信息在调试上下文中使用；没有它，oops列表只能以十六进制给你一个内核回溯，这不是很有用.

`CONFIG_IKCONFIG` `CONFIG_IKCONFIG_PROC`

- 这些选项（在“通用设置”菜单中找到）会导致完整的内核配置状态被构建到内核中，并通过/proc可用。大多数内核开发者知道他们使用的配置，并不需要这些选项（这会使内核变大）。然而，如果你正在尝试调试别人构建的内核中的问题，它们可能会有用。

`CONFIG_ACPI_DEBUG`

- 在“电源管理/ACPI”下。此选项打开详细的ACPI（高级配置和电源接口）调试信息，如果你怀疑与ACPI相关的问题，这可能会有用。

`CONFIG_DEBUG_DRIVER`

- 在“设备驱动程序”下。打开驱动程序核心的调试信息，这对于追踪低级支持代码中的问题可能有用。我们将在第14章中查看驱动程序核心。

`CONFIG_SCSI_CONSTANTS`

- 此选项，位于“设备驱动程序/SCSI设备支持”下，构建用于详细的SCSI错误消息的信息。如果你正在开发一个SCSI驱动程序，你可能需要这个选项。

`CONFIG_INPUT_EVBUG`

- 此选项（在“设备驱动程序/输入设备支持”下）打开输入事件的详细日志记录。如果你正在开发一个输入设备的驱动程序，这个选项可能会有帮助。但是，要注意这个选项的安全性问题：它记录你输入的所有内容，包括你的密码。

`CONFIG_PROFILING`

- 此选项在“性能分析支持”下找到。性能分析通常用于系统性能调优，但它也可以用于追踪一些内核挂起和相关问题。

我们将在查看追踪内核问题的各种方法时，重新讨论一些上述选项。但首先，我们将看看经典的调试技术：打印语句。

4.2. 用打印调试

最常见的调试技术是监视(monitoring)，这在应用程序编程中是通过在适当的点调用printf来完成的。当你在调试内核代码时，你可以用printk达到同样的目标。

4.2.1. printk

我们在前面的章节中使用了 `printk` 函数，简化的假设是它像printf一样工作。现在是时候介绍一些差异了。

其中一个差异是 `printk` 允许你根据消息的严重性分类消息，通过将不同的日志级别或优先级与消息关联起来。你通常用宏来表示日志级别。例如，我们在一些早期的打印语句中看到的 `KERN_INFO` ，就是消息可能的日志级别之一。日志级别宏扩展为一个字符串，该字符串在编译时与消息文本连接；这就是为什么在以下示例中，优先级和格式字符串之间没有逗号。以下是两个 `printk` 命令的示例，一个调试消息和一个关键消息：

```c
printk(KERN_DEBUG "Here I am: %s:%i\n", __FILE__, __LINE__);

printk(KERN_CRIT "I'm trashed; giving up on %p\n", ptr);
```

在头文件 `<linux/kernel.h>` 中定义了八个可能的日志级别字符串；我们按照严重性递减的顺序列出它们：

- `KERN_EMERG` 用于紧急消息，通常是崩溃之前的消息。
- `KERN_ALERT` 需要立即采取行动的情况。
- `KERN_CRIT` 关键条件，通常与严重的硬件或软件故障有关。
- `KERN_ERR` 用于报告错误条件；设备驱动程序经常使用KERN_ERR来报告硬件困难。
- `KERN_WARNING` 关于问题情况的警告，这些情况本身并不会对系统造成严重问题。
- `KERN_NOTICE` 情况是正常的，但仍值得注意。许多与安全相关的条件在这个级别报告。
- `KERN_INFO` 信息消息。许多驱动程序在启动时打印他们找到的硬件信息。
- `KERN_DEBUG` 用于调试消息。

每个字符串（在宏扩展中）代表一个尖括号中的整数。整数范围从0到7，较小的值代表较高的优先级.

没有指定优先级的 `printk` 语句默认为 `DEFAULT_MESSAGE_LOGLEVEL`，这在 `kernel/printk.c` 中被指定为一个整数。在2.6.10内核中，`DEFAULT_MESSAGE_LOGLEVEL` 是 `KERN_WARNING`，但在过去已经知道这可能会改变。

基于日志级别，内核可能将消息打印到当前控制台，无论是文本模式终端，串行端口，还是并行打印机。如果优先级小于整数变量 `console_loglevel`，消息会一行一行地发送到控制台（除非提供了尾随换行符，否则不会发送任何内容）。如果 `klogd` 和 `syslogd` 都在系统上运行，内核消息会被追加到 `/var/log/messages`（或者根据你的 `syslogd` 配置进行其他处理），这与 `console_loglevel` 无关。如果 `klogd` 没有运行，除非你读取 `/proc/kmsg`（通常最容易用 `dmesg` 命令完成），否则消息不会到达用户空间。

使用 klogd 时，你应该记住它不会保存连续的相同行；它只保存第一行，然后在稍后的时间，它接收到的重复次数。

变量 console_loglevel 被初始化为 DEFAULT_CONSOLE_LOGLEVEL，并可以通过 sys_syslog 系统调用进行修改。改变它的一种方法是在调用 klogd 时指定 -c 开关，如 klogd manpage 中所指定。注意，要改变当前值，你必须先杀掉 klogd，然后用 -c 选项重新启动它。或者，你可以编写一个程序来改变控制台日志级别。你会在O'Reilly的FTP站点提供的源文件的misc-progs/setlevel.c中找到这样一个程序的版本。新的级别被指定为1到8之间的整数值，包括1和8。如果设置为1，只有级别0（KERN_EMERG）的消息才能到达控制台；如果设置为8，所有消息，包括调试消息，都会显示。

也可以通过文本文件 /proc/sys/kernel/printk 读取和修改控制台日志级别。该文件包含四个整数值：当前日志级别，缺少明确日志级别的消息的默认级别，允许的最小日志级别，和启动时的默认日志级别。向这个文件写入一个值会将当前的日志级别改变为那个值；因此，例如，你可以通过简单地输入以下命令使所有内核消息在控制台上显示：

```shell
# echo 8 > /proc/sys/kernel/printk
```

现在应该很明显为什么hello.c样本有 KERN_ALERT ;标记；它们在那里是为了确保消息出现在控制台上。

- klogd 和 syslogd 都是Linux系统中用于处理和管理日志的守护进程。
- klogd 是内核日志守护进程，负责收集内核日志信息。它从/proc/kmsg设备文件读取内核消息，并将这些消息传递给 syslogd 进行处理。
- syslogd 是系统日志守护进程，负责处理来自各种系统应用的日志信息。它接收来自 klogd 的内核消息，以及其他系统服务和应用程序的日志消息，然后根据配置将这些消息写入到不同的日志文件中，或者转发给其他的日志服务器。
- 这两个守护进程通常一起工作，共同完成Linux系统的日志管理工作。

4.2.2. 重定向控制台消息

Linux通过允许你将消息发送到特定的虚拟控制台（如果你的控制台位于文本屏幕上）来提供一些在控制台日志策略上的灵活性。默认情况下，“控制台”是当前的虚拟终端。要选择不同的虚拟终端接收消息，你可以在任何控制台设备上发出ioctl(TIOCLINUX)。以下程序，setconsole，可以用来选择哪个控制台接收内核消息；它必须由超级用户运行，并且在

misc-progs目录中可用。以下是完整的程序。你应该用一个参数来调用它，指定要接收消息的控制台的编号。

```c
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/ioctl.h>

#include <errno.h>

#include <string.h>

int main(int argc, char **argv)

{
        //定义了一个字符数组，用于存储ioctl命令的参数
    char bytes[2] = {11,0}; /* 11 is the TIOCLINUX cmd
number */
        //如果命令行参数的数量为2（程序名和一个参数），则将第二个参数
        //argv[1]转换为整数，并存储在bytes[1]中.
    if (argc == 2) bytes[1] = atoi(argv[1]); /* the chosen
console */

    else {
                //如果命令行参数的数量不为2，程序将错误信息输出到标准
错误，并退出。
        fprintf(stderr, "%s: need a single arg\n",argv[0]);
exit(1);

    }
        //调用ioctl函数，使用TIOCLINUX命令，参数为`bytes`。如果调
用失败（返回值小于0），则输出错误信息并退出。
        //`ioctl`函数用于发送一个TIOCLINUX请求到标准输入设备（由
`STDIN_FILENO`指定）。这个请求用于设置哪个虚拟控制台接收内核消息。
    if (ioctl(STDIN_FILENO, TIOCLINUX, bytes)<0) {    /* use
stdin */

        fprintf(stderr,"%s: ioctl(stdin, TIOCLINUX): %s\n",

                argv[0], strerror(errno));
```

```
            exit(1);
        }
        exit(0);
    }
```

`setconsole`使用了特殊的`ioctl`命令`TIOCLINUX`，它实现了Linux特定的功能。要使用`TIOCLINUX`，你需要传递一个参数，这个参数是一个指向字节数组的指针。数组的第一个字节是一个数字，指定了请求的子命令，后面的字节是子命令特定的。在`setconsole`中，使用了子命令11，下一个字节（存储在`bytes[1]`中）标识了虚拟控制台。`TIOCLINUX`的完整描述可以在内核源码的`drivers/char/tty_io.c`中找到。

- `ioctl`函数是UNIX和Linux系统中的一个系统调用，用于设备特定的输入/输出操作。它的名称代表的是"input/output control"，即输入/输出控制。
- `ioctl`函数的原型如下：

```c
int ioctl(int fd, unsigned long request, ...);
```

- 这个函数接受三个参数：
- `fd`：这是一个文件描述符，通常是一个打开的设备文件。
- `request`：这是一个设备特定的请求码，用于指定要执行的操作。
- `...`：这是一个可选的参数，取决于`request`的值。这个参数通常是一个指针，指向一个数据结构，这个数据结构包含了操作所需的数据。
- `ioctl`函数的返回值取决于具体的请求。如果成功，它通常返回0。如果失败，它返回-1，并设置`errno`以指示错误。

4.2.3. 消息是如何记录的

printk函数将消息写入一个长度为 `__LOG_BUF_LEN` 字节的环形缓冲区，这个值在配置内核时可以选择从4KB到1MB。然后，该函数唤醒任何等待消息的进程，也就是说，任何在syslog系统调用中睡眠或正在读取/proc/kmsg的进程。这两个接口对于日志引擎来说几乎是等价的，但请注意，从/proc/kmsg读取会消耗日志缓冲区的数据，而syslog系统调用可以选择返回日志数据，同时保留给其他进程。通常来说，读取/proc文件更简单，这也是klogd的默认行为。dmesg命令可以用来查看缓冢区的内容，而不会清空它；实际上，该命令会将缓冲区的全部内容返回到stdout，无论它是否已经被读取。

如果你手动读取内核消息，在停止klogd后，你会发现/proc文件像FIFO一样，读取者会阻塞，等待更多的数据。显然，如果klogd或其他进程已经在读取相同的数据，你就不能这样读取消息，因为你会与它们争夺数据。

如果环形缓冲区满了，printk会回绕并开始向缓冲区的开始添加新数据，覆盖最旧的数据。因此，日志进程会丢失最旧的数据。但与使用这样的环形缓冲区的优点相比，这个问题可以忽略不计。例如，环形缓冲区允许系统即使在没有日志进程的情况下也能运行，同时通过覆盖旧数据来最小化内存浪费，如果没有人读取它的话。Linux处理消息的另一个特点是，printk可以从任何地方调用，甚至可以从中断处理程序中调用，没有打印数据的限制。唯一的缺点是可能会丢失一些数据。

如果klogd进程正在运行，它会获取内核消息并将它们发送给syslogd。然后，syslogd会检查/etc/syslog.conf来决定如何处理这些消息。syslogd会根据设施和优先级来区分消息；设施和优先级的可允许值都在<sys/syslog.h>中定义。内核消息通过LOG_KERN设施以与printk中使用的优先级相对应的优先级进行记录（例如，KERN_ERR消息使用LOG_ERR）。如果klogd没有运行，数据会保留在循环缓冲区中，直到有人读取它或者缓冲区溢出。

如果你不想让你的驱动程序的监控消息淹没你的系统日志，你可以选择两种方法。一种是给klogd指定-f（文件）选项，让它将消息保存到特定的文件中。另一种是自定义/etc/syslog.conf文件以满足你的需求。

4.2.4. 打开和关闭消息

在驱动程序开发的早期阶段，printk可以在调试和测试新代码方面提供很大的帮助。然而，当你正式发布驱动程序时，你应该移除或至少禁用这些打印语句。不幸的是，你可能会发现，一旦你认为你不再需要这些消息并移除它们，你在驱动程序中实现了一个新的功能（或者有人发现了一个bug），你想要重新开启至少一条消息。有几种方法可以解决这两个问题，全局启用或禁用你的调试消息，以及开启或关闭单个消息。

这里我们展示了一种编码printk调用的方式，你可以单独或全局地开启或关闭它们；这种技术依赖于定义一个宏，当你需要时，它可以解析为一个printk（或printf）调用：

- 每个打印语句可以通过移除或添加一个字母到宏的名字来启用或禁用。
- 所有的消息可以通过在编译前改变CFLAGS变量的值一次性禁用。

- 相同的打印语句可以在内核代码和用户级代码中使用，这样驱动程序和测试程序可以在额外消息方面以相同的方式管理。

以下代码片段实现了这些功能，并直接来自头文件scull.h：

```c
#undef PDEBUG             /* 无论如何，都取消定义它 */

#ifdef SCULL_DEBUG

#  ifdef __KERNEL__

     /* 如果开启了调试，并且在内核空间 */

#    define PDEBUG(fmt, args...) printk( KERN_DEBUG "scull:
" fmt, ## args)

#  else

     /* 这个用于用户空间 */

#    define PDEBUG(fmt, args...) fprintf(stderr, fmt, ##
args)

#  endif

#else

#  define PDEBUG(fmt, args...) /* 不调试: 什么都不做 */

#endif

#undef PDEBUGG

#define PDEBUGG(fmt, args...) /* 什么都不做: 它是一个占位符 */
```

1. `fmt`：这是一个格式字符串，类似于你在`printf`或`fprintf`函数中使用的那种。它定义了输出的格式。
2. `args...`：这是一个可变参数列表，表示这个宏可以接受任意数量的参数。这些参数将被插入到`fmt`指定的位置。

- 例如，如果你写`PDEBUG("value: %d", value)`，那么预处理器会将其替换为`printk( KERN_DEBUG "scull: value: %d", value)`。

- `## args`是GCC的一个扩展，用于处理没有参数的情况。如果没有提供参数，那么它会移除前面的逗号。例如，如果你写`PDEBUG("no args")`，那么预处理器会将其替换为`printk( KERN_DEBUG "scull: no args")`，而不是`printk( KERN_DEBUG "scull: no args", )`。

PDEBUG符号根据SCULL_DEBUG是否定义来定义或取消定义，并以适合代码运行环境的方式显示信息：当它在内核中时，使用内核调用printk，当在用户空间运行时，使用libc调用fprintf向标准错误输出。另一方面，PDEBUGG符号什么都不做；它可以用来轻松地“注释”打印语句，而不完全移除它们。

为了进一步简化过程，将以下行添加到你的makefile中：

```c
# 注释/取消注释以下行以禁用/启用调试

DEBUG = y

# 将你的调试标志（或不）添加到CFLAGS

ifeq ($(DEBUG),y)

  DEBFLAGS = -O -g -DSCULL_DEBUG # "-O" 是需要展开内联的

else

  DEBFLAGS = -O2

endif

#`CFLAGS += $(DEBFLAGS)`这行代码的效果是将`-O -g -DSCULL_DEBUG`这些编译器选项添加到`CFLAGS`变量中。然后，在makefile的其他地方，`CFLAGS`变量通常会被用在编译命令中，如`gcc $(CFLAGS) myfile.c`，这样这些选项就会被传递给编译器。

CFLAGS += $(DEBFLAGS)
```

1. `-O`：这是优化选项。它告诉编译器进行优化，但不指定优化级别。默认情况下，这将启用一些优化，但不会过分影响编译时间和生成的代码的可调试性。
2. `-g`：这是调试选项。它告诉编译器生成调试信息，这些信息可以被调试器（如gdb）使用。
3. `-DSCULL_DEBUG`：这是预处理器选项。它告诉预处理器定义一个名为 `SCULL_DEBUG` 的宏。在这个例子中，这个宏没有关联的值，所以它只是一个标志，用于在代码中检查是否定义了 `SCULL_DEBUG`。

本节中显示的宏依赖于gcc对ANSI C预处理器的扩展，该扩展支持具有可变数量参数的宏。这个gcc依赖性不应该是问题，因为内核本身就严重依赖于gcc的特性。此外，makefile依赖于GNU的make版本；再次，内核已经依赖于GNU make，所以这个依赖性不是问题。

如果你熟悉C预处理器，你可以在给定的定义上进行扩展，实现“调试级别”的概念，定义不同的级别，并为每个级别分配一个整数（或位掩码）值，以确定它应该有多详细。

但是，每个驱动程序都有自己的特性和监控需求。良好编程的艺术在于选择灵活性和效率之间的最佳折衷，我们无法告诉你什么是最好的。记住，预处理器条件（以及代码中的常量表达式）是在编译时执行的，所以你必须重新编译以开启或关闭消息。一个可能的替代方案是使用C条件，它们在运行时执行，因此，允许你在程序执行期间开启和关闭消息。这是一个很好的特性，但是它需要每次执行代码时进行额外的处理，即使消息被禁用，也会影响性能。有时候，这种性能损失是无法接受的。

本节中显示的宏在许多情况下都已经证明是有用的，唯一的缺点是在对其消息进行任何更改后，需要重新编译模块。

4.2.5. 速率限制

如果不小心，你可能会发现自己生成了数千条printk消息，这可能会淹没控制台，甚至可能溢出系统日志文件。当使用一个慢速的控制台设备（例如，串行端口）时，过高的消息速率也可能会减慢系统的运行速度，或者使其无响应。当控制台不停地输出数据时，很难找出系统出了什么问题。因此，你应该非常小心你打印的内容，特别是在驱动程序的生产版本中，特别是在初始化完成后。一般来说，生产代码在正常操作过程中不应打印任何东西；打印输出应该是需要注意的异常情况的指示。

另一方面，如果你正在驱动的设备停止工作，你可能想要发出一个日志消息。但是你应该小心不要过度做事。一个在失败面前无休止地继续的不智能的过程可能每秒生成数千次重试；如果你的驱动程序每次都打印一个“我的设备坏了”的消息，它可能会产生大量的输出，如果控制台设备慢的话，可能会占用CPU——即使是串行端口或行打印机，也不能使用中断来驱动控制台。

在许多情况下，最好的行为是设置一个标志，表示“我已经抱怨过这个问题了”，并且一旦标志被设置，就不再打印任何进一步的消息。然而，在其他情况下，有理由偶尔发出一个“设备

仍然坏了”的通知。

内核提供了一个函数，可以在这种情况下有所帮助：
`int printk_ratelimit(void);`
在你考虑打印一个可能会被经常重复的消息之前，应该调用这个函数。如果函数返回一个非零值，那么就打印你的消息，否则就跳过它。因此，典型的调用看起来像这样：

```c
if (printk_ratelimit( ))
        printk(KERN_NOTICE "The printer is still on
fire\n");
```

printk_ratelimit通过跟踪发送到控制台的消息数量来工作。当输出的级别超过一个阈值时，printk_ratelimit开始返回0，并导致消息被丢弃。

可以通过修改/proc/sys/kernel/printk_ratelimit（在重新启用消息之前等待的秒数）和/proc/sys/kernel/printk_ratelimit_burst（在限制速率之前接受的消息数量）来定制printk_ratelimit的行为。

4.2.6. 打印设备编号

偶尔，当从驱动程序打印消息时，你可能会想打印与感兴趣的硬件相关联的设备号。打印主要和次要数字并不特别困难，但是，出于一致性的考虑，内核提供了一对实用宏（在<linux/kdev_t.h>中定义）来完成这个任务：

```c
int print_dev_t(char *buffer, dev_t dev);
char *format_dev_t(char *buffer, dev_t dev);
```

这两个宏都将设备号编码到给定的缓冲区；唯一的区别是print_dev_t返回打印的字符数，而format_dev_t返回缓冲区；因此，它可以直接作为一个参数用于printk调用，尽管必须记住，printk在提供尾随换行符之前不会刷新。缓冲区应该足够大以容纳设备号；考虑到64位设备号在未来的内核版本中是一个明显的可能性，缓冲区可能应该至少有20字节长。

4.3. 用查询来调试

前一部分描述了printk的工作方式以及如何使用它，但并未讨论其缺点。

大量使用printk会明显地减慢系统的速度，即使你降低console_loglevel以避免加载控制台设备，因为syslogd会持续同步其输出文件；因此，每打印一行就会导致一次磁盘操作。从syslogd的角度来看，这是正确的实现。它试图将所有内容尽快写入磁盘，以防系统在打印消息后立即崩溃；然而，你并不希望仅仅为了调试消息就减慢你的系统。这个问题可以通过在/etc/syslogd.conf中出现的日志文件名前加上连字符来解决。修改配置文件的问题在于，即使你完成了调试，修改可能仍会存在，尽管在正常系统操作中，你希望消息尽快被刷新到磁盘。除了这种永久性的改变，还有一种替代方案是运行一个不同于klogd的程序（如前面建议的cat /proc/kmsg），但这可能无法为正常的系统操作提供适当的环境。

通常，获取相关信息的最佳方式是在需要信息时查询系统，而不是持续产生数据。实际上，每个Unix系统都提供了许多获取系统信息的工具：ps，netstat，vmstat等。

驱动开发者有几种查询系统的技术：在/proc文件系统中创建一个文件，使用ioctl驱动方法，以及通过sysfs导出属性。使用sysfs需要对驱动模型有相当多的了解。这在第14章中进行了讨论。

4.3.1. 使用 /proc 文件系统

/proc文件系统是一个特殊的，由软件创建的文件系统，被内核用来向外界导出信息。/proc下的每个文件都与一个内核函数绑定，该函数在读取文件时即时生成文件的“内容”。我们已经看到了一些这样的文件在操作中；例如，/proc/modules总是返回当前加载的模块的列表。

/proc在Linux系统中被大量使用。现代Linux发行版上的许多实用程序，如ps，top和uptime，都从/proc获取信息。一些设备驱动程序也通过/proc导出信息，你的设备也可以这样做。/proc文件系统是动态的，所以你的模块可以随时添加或删除条目。

完全功能的/proc条目可能是复杂的野兽；它们可以被写入以及读取。然而，大多数时候，/proc条目是只读文件。这一部分关注的是简单的只读情况。对于那些有兴趣实现更复杂的东西的人，可以在这里查看基础知识；然后可以查阅内核源码以获取完整的信息。

然而，在我们继续之前，我们应该提到，在/proc下添加文件是不被鼓励的。内核开发者认为/proc文件系统是一种无法控制的混乱，已经远远超出了其原始目的（即提供关于系统中正在运行的进程的信息）。在新代码中提供信息的推荐方式是通过sysfs。然而，如前所述，使用sysfs需要理解Linux设备模型，我们直到第14章才会讲到这个。与此同时，/proc下的文件稍微容易创建一些，它们完全适合于调试目的，所以我们在这里介绍它们。

4.3.1.1. 在 /proc 里实现文件

所有与/proc一起工作的模块都应该包含<linux/proc_fs.h>以定义适当的函数。

要创建一个只读的/proc文件，你的驱动程序必须实现一个函数，当文件被读取时产生数据。当某个进程读取文件（使用read系统调用）时，请求通过这个函数到达你的模块。我们将首先看看这个函数，然后在本节后面看看注册接口。

当一个进程从你的/proc文件中读取时，内核分配了一块内存页（即，PAGE_SIZE字节），驱动程序可以在其中写入要返回给用户空间的数据。该缓冲区被传递给你的函数，这是一个名为read_proc的方法：

```c
int (*read_proc)(char *page, char **start, off_t offset, int
count,
                 int *eof, void *data);
```

page指针是你将写入数据的缓冲区；start被函数用来表示在page中已经写入了的数据的位置（稍后会有更多解释）；offset和count的含义与read方法相同。eof参数指向一个必须由驱动程序设置的整数，用来表示它没有更多的数据要返回，而data是你可以用于内部记账的驱动程序特定的数据指针。

这个函数应该返回实际放入page缓冲区的数据的字节数，就像其他文件的read方法一样。其他的输出值是 `_eof` 和 `_start`。eof是一个简单的标志，但是start值的使用稍微复杂一些；它的目的是帮助实现大于一页的/proc文件。

start参数有一个有些非传统的用途。它的目的是指示在哪里（在page内）找到要返回给用户的数据。当你的proc_read方法被调用时，`_start` 将为NULL。如果你让它保持为NULL，内核假设数据已经被放入page，就像offset为0一样；换句话说，它假设一个简单的proc_read版本，将虚拟文件的全部内容放入page，而不考虑offset参数。如果你将 `_start` 设置为非NULL值，内核假设 `_start` 指向的数据已经考虑了offset，并且准备直接返回给用户。通常，简单的proc_read方法返回微小的数据量就忽略start。更复杂的方法将 `_start` 设置为page，并且只在请求的offset处开始放置数据。

长期以来，/proc文件还有另一个主要问题，start也是为了解决这个问题。有时，内核数据结构的ASCII表示在连续的read调用之间会改变，所以读取进程可能会发现从一个调用到下一个调用的数据不一致。如果 `_start` 被设置为一个小的整数值，调用者使用它来独立于你返回的数据量增加filp→f_pos，从而使f_pos成为你的read_proc过程的内部记录号。例如，如果你的read_proc函数正在返回来自一个大数组的结构的信息，并且在第一次调用中返回了五个这样的结构，`_start` 可以被设置为5。下一次调用提供了相同的值作为偏移量；驱动程序然后知道从数组中的第六个结构开始返回数据。这被它的作者们承认是一个“hack”，可以在fs/proc/generic.c中看到。

注意，有一种更好的方法来实现大的/proc文件；它被称为seq_file，我们稍后会讨论它。首先，是时候给出一个例子了。这是一个简单的（如果有些丑陋的）scull设备的read_proc实现：

```c
int scull_read_procmem(char *buf, char **start, off_t
offset,
                   int count, int *eof, void *data)
{
    int i, j, len = 0;
    int limit = count - 80; /* Don't print more than this */
    for (i = 0; i < scull_nr_devs && len <= limit; i++) {
        struct scull_dev *d = &scull_devices[i];
        struct scull_qset *qs = d->data;
        if (down_interruptible(&d->sem))
            return -ERESTARTSYS;
        len += sprintf(buf+len,"\nDevice %i: qset %i, q %i,
sz %li\n",i, d->qset, d->quantum, d->size);
        for (; qs && len <= limit; qs = qs->next) { /* scan
the list */
            len += sprintf(buf + len, "  item at %p, qset at
%p\n",
                       qs, qs->data);
            if (qs->data && !qs->next) /* dump only the last
item */
                for (j = 0; j < d->qset; j++) {
                    if (qs->data[j])
                        len += sprintf(buf + len,
```

```
                                "    % 4i: %8p\n",
                                j, qs->data[j]);
                    }
            }
            up(&scull_devices[i].sem);
    }
    *eof = 1;
    return len;
}
```

这是一个相当典型的read_proc实现。它假设永远不需要生成超过一页的数据，所以忽略了start和offset值。然而，它小心不要溢出它的缓冲区，以防万一。

- `sprintf`函数是一个格式化字符串的函数，它将格式化的结果写入给定的字符串中。
- 在这个例子中，格式化的字符串是"\nDevice %i: qset %i, q %i, sz %li\n"，其中的%i和%li是格式说明符，它们表示将被替换为相应的参数值。
- 第一个%i被替换为变量i的值，表示设备的编号。
- 第二个%i被替换为d->qset的值，表示设备的qset值。
- 第三个%i被替换为d->quantum的值，表示设备的quantum值。
- %li被替换为d->size的值，表示设备的大小。
- `buf+len`是指向缓冲区中的某个位置的指针，这个位置是当前已经写入的数据的末尾。这样，新的数据就会被添加到已有数据的后面，而不是覆盖已有数据。

- `len += sprintf(...)`是在更新len的值，使其包含新添加的数据的长度。这样，len就始终表示缓冲区中的数据的总长度。

`sprintf`是C语言中的一个函数，它用于将格式化的数据写入字符串。它的原型如下：

int sprintf(char *str, const char* format, ...);

- `str`：这是指向一个字符数组的指针，`sprintf`将把结果字符串写入这个数组。
- `format`：这是一个格式字符串，它可以包含一些格式说明符，如`%d`（表示整数）、`%f`（表示浮点数）、`%s`（表示字符串）等。格式字符串中的每个格式说明符都对应于一个额外的参数，这个参数的值将被格式化并插入到结果字符串中。
- `...`：这是一系列额外的参数，它们的类型和数量应该与格式字符串中的格式说明符相匹配。

`sprintf`函数返回的是被写入`str`的字符数（不包括最后的空字符）。

例如，下面的代码：

```c
char buf[100];

int a = 5;

float b = 3.14;

sprintf(buf, "a = %d, b = %f", a, b);
```

将会把字符串"a = 5, b = 3.140000"写入`buf`数组。

4.3.1.2. 老接口

如果你阅览内核源码, 你会遇到使用老接口实现 /proc 的代码:

```c
int (*get_info)(char *page, char **start, off_t offset, int
count);
```

所有的参数的含义同 read_proc 的相同, 但是没有 eof 和 data 参数. 这个接口仍然支持, 但是将来会消失; 新代码应当使用 read_proc 接口来代替.

4.3.1.3. 创建你的 /proc 文件

一旦你定义了一个read_proc函数，你需要将它连接到/proc层次结构中的一个条目。这是通过调用create_proc_read_entry来完成的：

```c
struct proc_dir_entry *create_proc_read_entry(const char
*name,mode_t mode, struct proc_dir_entry *base,read_proc_t
*read_proc, void *data);
```

这里，name是要创建的文件的名称，mode是文件的保护掩码（可以传递0作为系统默认），base指示应创建文件的目录（如果base为NULL，文件将在/proc根目录下创建），read_proc是实现文件的read_proc函数，data被内核忽略（但传递给read_proc）。这是scull使用的调用，使其/proc函数可用为/proc/scullmem：

```c
create_proc_read_entry("scullmem", 0 /* default mode */,
        NULL /* parent dir */, scull_read_procmem,
        NULL /* client data */);
```

这里，我们在/proc下直接创建了一个名为scullmem的文件，具有默认的、全世界可读的保护。

目录条目(entry)指针可以用来在/proc下创建整个目录层次结构。然而，注意，一个条目可能更容易地放在/proc的子目录中，只需将目录名作为条目名的一部分，只要目录本身已经存在。例如，一个（经常被忽视的）约定说，与设备驱动程序相关的/proc条目应该放在driver/子目录中；scull可以通过将其名称设为driver/scullmem来将其条目放在那里。

当模块被卸载时，应该删除/proc中的条目。remove_proc_entry是撤销create_proc_read_entry已经做的事情的函数：

```c
remove_proc_entry("scullmem", NULL /* parent dir */);
```

如果不删除条目，可能会在不需要的时候调用，或者，如果你的模块已经被卸载，可能会导致内核崩溃。

当使用/proc文件时，你必须记住实现的一些麻烦事——这也不奇怪，现在不鼓励使用它。

最重要的问题是删除/proc条目。这样的删除可能会在文件正在使用时发生，因为没有所有者与/proc条目关联，所以使用它们不会影响模块的引用计数。这个问题可以通过在移除模块之前运行sleep 100 < /proc/myfile来触发。

另一个问题是注册两个同名的条目。内核信任驱动程序，并不检查名称是否已经注册，所以如果你不小心，你可能会结束了两个或更多同名的条目。这是一个已知的问题，这样的条目在访问它们和调用remove_proc_entry时是无法区分的。

- "条目"（entry）通常指的是在文件系统中的一个文件或目录。在/proc文件系统中，每一个"条目"都对应一个文件或目录，这些文件或目录提供了一种方式来读取和写入内核数据。
- 你可以把"条目"理解为/proc文件系统中的一个接口，通过这个接口，用户空间的程序可以与内核空间的数据进行交互。

4.3.1.4. seq_file 接口

如我们之前所提到的，大文件在 /proc 下的实现有些尴尬。随着时间的推移，当输出量变大时，/proc 的方法因其错误的实现而臭名昭著。为了清理 /proc 的代码并使内核程序员的生活更轻松，添加了 `seq_file` 接口。这个接口为大型内核虚拟文件的实现提供了一套简单的函数。

`seq_file` 接口假设你正在创建一个虚拟文件，该文件通过一系列需要返回给用户空间的项目。要使用 `seq_file`，你必须创建一个简单的“迭代器”对象，该对象可以在序列中建立位置，向前步进，并输出序列中的一个项目。这可能听起来很复杂，但实际上，这个过程非常简单。我们将通过在 scull 驱动程序中创建一个 /proc 文件来展示如何完成这个过程。

- 迭代器是一种设计模式，它是一个对象，它可以遍历并选择序列中的对象，而开发人员不需要了解该序列的底层结构。迭代器提供了一种方法来访问集合对象中的元素，而又不暴露该对象的底层表示。

首先，必须包含 `<linux/seq_file.h>`。然后你必须创建四个迭代器方法，分别是 start、next、stop 和 show。

start 方法总是首先被调用。这个函数的原型是：

```
void *start(struct seq_file *sfile, loff_t *pos);
```

sfile 参数几乎总是被忽略。pos 是一个整数位置，表示读取应该从哪里开始。位置的解释完全取决于实现；它不必是结果文件中的字节位置。由于 seq_file 实现通常通过一系列有趣的项目，位置通常被解释为指向序列中下一个项目的光标。scull 驱动程序将每个设备解释为序列中的一个项目，所以传入的 pos 就是 scull_devices 数组的索引。因此，scull 中使用的 start 方法是：

```c
static void *scull_seq_start(struct seq_file *s, loff_t
*pos)

{

    if (*pos ≥ scull_nr_devs)

        return NULL;    /* No more to read */

    return scull_devices + *pos;

}
```

如果返回值非空，那么它就是迭代器实现可以使用的私有值。

next 函数应将迭代器移动到下一个位置，如果序列中没有剩余的元素，就返回 NULL。这个方法的原型是：

```
void *next(struct seq_file *sfile, void *v, loff_t *pos);
```

这里，v 是从前一次调用 start 或 next 返回的迭代器，pos 是文件中的当前位置。next 应该增加 pos 指向的值；根据你的迭代器如何工作，你可能（虽然可能不会）想要将 pos 增加多于一个的值。这是 scull 的实现：

```c
static void *scull_seq_next(struct seq_file *s, void *v,
loff_t *pos)

{

    (*pos)++;

    if (*pos ≥ scull_nr_devs)

        return NULL;

    return scull_devices + *pos;

}
```

当内核完成迭代器的使用，它会调用 stop 来进行清理：

```
void stop(struct seq_file *sfile, void *v);
```

scull 的实现没有清理工作要做，所以它的 stop 方法是空的。

值得注意的是，seq_file 代码的设计，不会在调用 start 和 stop 之间进行睡眠或执行其他非原子任务。你也可以保证在调用 start 后不久就会看到一次 stop 调用。因此，你的 start 方法可以安全地获取信号量或自旋锁。只要你的其他 seq_file 方法是原子的，整个调用序列就是原子的。（如果这段话你不理解，等你读完下一章再回来看。）

在这些调用之间，内核调用 show 方法实际向用户空间输出一些有趣的东西。这个方法的原型是：

```
int show(struct seq_file *sfile, void *v);
```

这个方法应该为迭代器 v 指示的序列中的项目创建输出。然而，它不应该使用 printk；相反，有一套专门的函数用于 seq_file 输出：

```
int seq_printf(struct seq_file *sfile, const char *fmt, ...);
```

这是 seq_file 实现的 printf 等价物；它接受通常的格式字符串和额外的值参数。然而，你也必须将给 show 函数的 seq_file 结构传递给它。如果 seq_printf 返回一个非零值，那么意味着缓冲区已满，输出正在被丢弃。然而，大多数实现都忽略了返回值。

```c
int seq_putc(struct seq_file *sfile, char c);

int seq_puts(struct seq_file *sfile, const char *s);
```

这些是用户空间 putc 和 puts 函数的等价物。

```c
int seq_escape(struct seq_file *m, const char *s, const char *esc)
;
```

这个函数等价于 seq_puts，除了在 s 中的任何字符也在 esc 中找到的情况下，它会以八进制格式打印。esc 的常见值是 " \t\n"，这可以防止嵌入的空白破坏输出并可能混淆 shell 脚本。

```c
int seq_path(struct seq_file *sfile, struct vfsmount *m,
struct dentry

  *dentry, char *esc);
```

这个函数可以用于输出与给定目录条目关联的文件名。它在设备驱动程序中不太可能有用；我们在这里包含它是为了完整性。

回到我们的例子；在 scull 中使用的 show 方法是：

```c
static int scull_seq_show(struct seq_file *s, void *v)
{
    struct scull_dev *dev = (struct scull_dev *) v;
    struct scull_qset *d;
    int i;
    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
    seq_printf(s, "\nDevice %i: qset %i, q %i, sz %li\n",
            (int) (dev - scull_devices), dev->qset,
            dev->quantum, dev->size);
    for (d = dev->data; d; d = d->next) { /* scan the list
*/
        seq_printf(s, "  item at %p, qset at %p\n", d, d-
>data);
        if (d->data && !d->next) /* dump only the last item
*/
            for (i = 0; i < dev->qset; i++) {
                if (d->data[i])
                    seq_printf(s, "    % 4i: %8p\n",
                            i, d->data[i]);
            }
    }
```

```
    up(&dev->sem);

    return 0;

}
```

在这里，我们最后解释了我们的“迭代器”值，它只是一个指向 scull_dev 结构的指针。

现在，scull 必须打包它的一整套迭代器操作，并将它们连接到 /proc 中的一个文件。第一步是通过填充一个 seq_operations 结构来完成的：

```c
static struct seq_operations scull_seq_ops = {

    .start = scull_seq_start,

    .next  = scull_seq_next,

    .stop  = scull_seq_stop,

    .show  = scull_seq_show

};
```

有了这个结构，我们必须创建一个内核能理解的文件实现。我们不使用前面描述的 read_proc 方法；当使用 seq_file 时，最好在 /proc 中稍微低一级的地方连接。这意味着创建一个 file_operations 结构（是的，这是用于字符驱动程序的相同结构）实现内核处理文件上的读取和搜索所需的所有操作。幸运的是，这个任务很简单。第一步是创建一个将文件连接到 seq_file 操作的 open 方法：

```c
static int scull_proc_open(struct inode *inode, struct file
*file)

{

    return seq_open(file, &scull_seq_ops);

}
```

调用 seq_open 将文件结构与我们上面定义的序列操作连接起来。事实证明，open 是我们必须自己实现的唯一文件操作，所以我们现在可以设置我们的 file_operations 结构：

```c
static struct file_operations scull_proc_ops = {

    .owner   = THIS_MODULE,

    .open    = scull_proc_open,

    .read    = seq_read,

    .llseek  = seq_lseek,

    .release = seq_release

};
```

在这里，我们指定了我们自己的 open 方法，但对于其他所有操作，我们使用了预设的方法 seq_read、seq_lseek 和 seq_release。

最后一步是在 /proc 中创建实际的文件：

```c
entry = create_proc_entry("scullseq", 0, NULL);
if (entry)
    entry->proc_fops = &scull_proc_ops;
```

我们调用了更低级别的 create_proc_entry，而不是使用 create_proc_read_entry，它的原型是：

```c
struct proc_dir_entry *create_proc_entry(const char *name,
                                         mode_t mode,
                                         struct proc_dir_entry
*parent);
```

参数与 create_proc_read_entry 中的等价物相同：文件的名称、它的保护和父目录。

1. 定义一个 `seq_operations` 结构，这个结构包含了四个函数指针，分别对应迭代器的四个操作：开始（start）、下一个（next）、停止（stop）和显示（show）。这四个函数在其他地方定义，用于控制如何遍历和显示 scull 设备的状态。
2. 定义一个 `file_operations` 结构，这个结构定义了内核如何处理文件操作。在这个结构中，`open` 方法被设置为 `scull_proc_open`，这个函数调用 `seq_open` 来将文件结构与 `seq_operations` 结构关联起来。其他的文件操作（读取、定位和释放）则使用了预设的 `seq_file` 操作。
3. 最后，使用 `create_proc_entry` 函数在 /proc 文件系统中创建一个名为 "scullseq" 的文件，并将其文件操作设置为 `scull_proc_ops`。

有了上面的代码，scull 就有了一个新的 /proc 条目，看起来很像之前的那个。然而，它更优越，因为无论它的输出变得多大，它都能正常工作，它能正确处理搜索，并且通常更容易阅读和维护。我们建议使用 seq_file 来实现包含超过很小数量的输出行的文件。

4.3.2. ioctl 方法

在第六章中，我们介绍了如何使用ioctl，这是一个作用于文件描述符的系统调用；它接收一个标识要执行的命令的数字和（可选的）另一个参数，通常是一个指针。作为使用/proc文件

系统的替代方案，你可以实现一些为调试定制的ioctl命令。这些命令可以将驱动程序中相关的数据结构复制到用户空间，以便你可以检查它们。

使用ioctl这种方式获取信息比使用/proc稍微困难一些，因为你需要另一个程序来发出ioctl并显示结果。这个程序必须被编写、编译，并与你正在测试的模块保持同步。

另一方面，驱动程序端的代码可能比实现/proc文件所需的代码更简单。

有时候，ioctl是获取信息的最佳方式，因为它的运行速度比读取/proc快。如果在数据写入屏幕之前需要对数据进行一些处理，那么以二进制形式获取数据比读取文本文件更有效率。此外，ioctl不需要将数据分割成小于一页的片段。

ioctl方法的另一个有趣的优点是，即使在其他情况下禁用了调试，也可以在驱动程序中保留信息检索命令。与/proc文件不同，后者对任何查看目录的人都是可见的（而且很可能有太多的人会想知道“那个奇怪的文件是什么”），未记录的ioctl命令可能会被忽视。此外，如果驱动程序发生了一些奇怪的事情，它们仍然会在那里。唯一的缺点是模块会稍微大一些。

4.4. 使用观察来调试

有时，通过观察用户空间应用程序的行为，可以追踪到一些小问题。观察程序也有助于建立对驱动程序正常工作的信心。例如，我们在观察scull的read实现如何对不同数据量的read请求做出反应后，对其感到满意。

有多种方法可以观察用户空间程序的工作。你可以在它上面运行调试器来逐步执行其函数，添加打印语句，或者在strace下运行程序。在这里，我们只讨论最后一种技术，当真正的目标是检查内核代码时，这种技术最有趣。

strace命令是一个强大的工具，它显示了用户空间程序发出的所有系统调用。它不仅显示调用，还可以以符号形式显示调用的参数和它们的返回值。当系统调用失败时，错误的符号值（例如，ENOMEM）和相应的字符串（内存不足）都会被显示出来。strace有许多命令行选项；最有用的是-t，用于显示每个调用执行的时间，-T，用于显示在调用中花费的时间，-e，用于限制跟踪的调用类型，和-o，用于将输出重定向到文件。默认情况下，strace在stderr上打印跟踪信息。

strace从内核本身接收信息。这意味着，无论程序是否使用调试支持（gcc的-g选项）编译，无论它是否被剥离，都可以对其进行跟踪。你也可以将跟踪附加到正在运行的进程，类似于调试器可以连接到正在运行的进程并控制它。

跟踪信息经常被用来支持发送给应用程序开发者的错误报告，但对内核程序员来说，它也是无价的。我们已经看到驱动程序代码是如何通过对系统调用的反应来执行的；strace允许我们检查每个调用的输入和输出数据的一致性。

下面的代码显示了运行命令`strace ls /dev > /dev/scull0`的最后几行（大部分）：

SHELL

```
open("/dev", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY) =
3

fstat64(3, {st_mode=S_IFDIR|0755, st_size=24576, ...}) = 0

fcntl64(3, F_SETFD, FD_CLOEXEC)         = 0

getdents64(3, /* 141 entries */, 4096)  = 4088

[...]

getdents64(3, /* 0 entries */, 4096)    = 0

close(3)                                = 0

[...]

fstat64(1, {st_mode=S_IFCHR|0664, st_rdev=makedev(254, 0),
...}) = 0

write(1, "MAKEDEV\nadmmidi0\nadmmidi1\nadmmid"..., 4096) =
4000

write(1, "b\nptywc\nptywd\nptywe\nptywf\nptyx0\n"..., 96) =
96

write(1, "b\nptyxc\nptyxd\nptyxe\nptyxf\nptyy0\n"..., 4096)
= 3904

write(1, "s17\nvcs18\nvcs19\nvcs2\nvcs20\nvcs21"..., 192) =
192

write(1, "\nvcs47\nvcs48\nvcs49\nvcs5\nvcs50\nvc"..., 673) =
673

close(1)                                = 0

exit_group(0)                           = ?
```

从第一个write调用可以看出，ls在完成目标目录的查看后，试图写入4KB。奇怪的是（对于ls来说），只写入了4000字节，然后操作被重试。然而，我们知道scull中的write实现一次只写入一个量子，所以我们可能预期到部分写入。经过几步后，一切都顺利进行，程序成功退出。

这个例子展示了如何使用strace来观察用户空间程序与内核交互的细节，这对于理解和调试驱动程序的行为非常有用。

作为另一个例子，让我们读取scull设备（使用wc命令）：

```c
[...]

open("/dev/scull0", O_RDONLY|O_LARGEFILE) = 3

fstat64(3, {st_mode=S_IFCHR|0664, st_rdev=makedev(254, 0),
...}) = 0

read(3, "MAKEDEV\nadmmidi0\nadmmidi1\nadmmid"..., 16384) =
4000

read(3, "b\nptywc\nptywd\nptywe\nptywf\nptyx0\n"..., 16384)
= 4000

read(3, "s17\nvcs18\nvcs19\nvcs2\nvcs20\nvcs21"..., 16384) =
865

read(3, "", 16384)                      = 0

fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1),
...}) = 0

write(1, "8865 /dev/scull0\n", 17)      = 17

close(3)                                = 0

exit_group(0)                           = ?
```

如预期的那样，read一次只能检索4000字节，但总数据量与前一个例子中写入的数据量相同。值得注意的是，这个例子中的重试是如何组织的，与前一个跟踪相反。wc被优化为快速

读取，因此，它绕过了标准库，试图用一个系统调用读取更多的数据。你可以从跟踪中的read行看到，wc试图一次读取16KB。

Linux专家可以在strace的输出中找到许多有用的信息。如果你被所有的符号吓到了，你可以限制自己观察文件方法（如open，read等）如何使用efile标志工作。

就我们个人而言，我们发现strace在定位系统调用的运行时错误方面最有用。通常，应用程序或演示程序中的perror调用不够详细，无法用于调试，而能够准确地知道哪些参数触发了哪个系统调用的错误，这会是很大的帮助。

- strace是用来跟踪用户空间进程的系统调用和信号的.
- 系统调用是指**运行在用户空间的程序向操作系统内核请求需要更高权限运行的服务。**
- 系统调用提供用户程序与操作系统之间的接口.
- 内核提供给用户空间的这些API, 就是系统调用.
- `strace`是一个强大的命令行工具，它可以跟踪到一个进程执行的系统调用和接收的信号。在调试内核代码中的运行时错误时，`strace`可以提供以下帮助：

1. **查看系统调用的详细信息**：`strace`可以显示出一个进程所有的系统调用，包括调用的参数，返回值，以及可能的错误信息。这对于理解和调试内核代码的行为非常有用。
2. **查看系统调用的执行顺序**：`strace`按照执行顺序显示系统调用，这可以帮助我们理解程序的执行流程，找出可能的问题。
3. **查看系统调用的错误信息**：如果一个系统调用失败了，`strace`会显示出错误信息。这对于找出运行时错误的原因非常有用。
4. **查看进程的信号交互**：`strace`还可以显示出进程接收和发送的信号，这对于理解和调试进程间的通信非常有用。

- 使用`strace`的一个常见场景是，当你的程序异常退出或者行为不符合预期时，你可以使用`strace`来查看程序在运行过程中进行了哪些系统调用，以及这些系统调用的结果如何，从而找出可能的问题所在。

- 例如，如果你的程序在读取一个文件时失败了，你可以使用`strace`来查看`open`，`read`等系统调用的详细信息，看看是不是文件路径错误，或者权限不足等问题。

4.5. 调试系统故障

即使你已经使用了所有的监控和调试技术，有时候驱动程序中仍然存在错误，当驱动程序被执行时，系统会出现故障。当这种情况发生时，收集尽可能多的信息以解决问题是非常重要的。

注意，“故障fault”并不意味着“恐慌panic”。Linux代码足够健壮robust，能够优雅地应对大多数错误：一个故障通常会导致当前进程的销毁，而系统继续运行。系统可以进入恐慌状态，如果故障发生在进程的上下文之外，或者系统的某个重要部分被破坏，系统可能会进入恐慌状态。但是，当问题是由于驱动程序错误引起的，通常只会导致使用该驱动程序的进程

突然死亡。当一个进程被销毁时，唯一无法恢复的损害是一些分配给进程上下文的内存丢失；例如，驱动程序通过kmalloc分配的动态列表可能会丢失。然而，由于内核在进程死亡时会调用任何打开设备的关闭操作，所以你的驱动程序可以释放由打开方法分配的内容。

尽管一个oops通常不会导致整个系统崩溃，但你可能会发现自己在一个oops发生后需要重启。一个有错误的驱动程序可能会使硬件处于无法使用的状态，使内核资源处于不一致的状态，或者在最坏的情况下，随机地破坏内核内存。通常，你可以简单地卸载你的有问题的驱动程序，然后在oops之后再试一次。然而，如果你看到任何表明整个系统不太好的东西，你最好的选择通常是立即重启。

我们已经说过，当内核代码行为不当时，会在控制台上打印出一条信息。下一节将解释如何解码和使用这些信息。尽管它们对新手来说看起来相当模糊，但处理器的转储信息充满了有趣的信息，通常足以定位程序错误，而无需进行额外的测试。

4.5.1. oops 消息

大多数错误都表现为NULL指针解引用或使用其他不正确的指针值。这类错误的常见结果是oops消息。

处理器使用的几乎所有地址都是虚拟地址，通过复杂的页表结构映射到物理地址（例外情况是与内存管理子系统本身一起使用的物理地址）。当一个无效的指针被解引用时，分页机制无法将指针映射到物理地址，处理器向操作系统发出页错误信号。如果地址无效，内核无法“页入”缺失的地址；如果在处理器处于监督模式时发生这种情况，它（通常）会生成一个oops。

一个oops显示了故障时的处理器状态，包括CPU寄存器的内容和其他看似无法理解的信息。这个消息是由故障处理器 `(arch/*/kernel/traps.c)` 中的printk语句生成的，并按照前面“printk”部分所述的方式分派。

让我们看一个这样的消息。这是在运行内核版本2.6的PC上解引用NULL指针的结果。这里最相关的信息是指令指针（EIP），即错误指令的地址。

```shell
Unable to handle kernel NULL pointer dereference at virtual
address 00000000
 printing eip:
d083a064
Oops: 0002 [#1]
SMP
CPU:  0
EIP:  0060:[<d083a064>]  Not tainted
EFLAGS: 00010246  (2.6.6)
EIP is at faulty_write+0x4/0x10 [faulty]
eax: 00000000  ebx: 00000000  ecx: 00000000  edx: 00000000
esi: cf8b2460  edi: cf8b2480  ebp: 00000005  esp: c31c5f74
ds: 007b  es: 007b  ss: 0068

Process bash (pid: 2086, threadinfo=c31c4000 task=cfa0a6c0)
Stack: c0150558 cf8b2460 080e9408 00000005 cf8b2480 00000000
cf8b2460 cf8b2460 fffffff7 080e9408 c31c4000 c0150682
cf8b2460 080e9408 00000005 cf8b2480 00000000 00000001
00000005 c0103f8f 00000001 080e9408 00000005 00000005
Call Trace:
 [<c0150558>] vfs_write+0xb8/0x130
 [<c0150682>] sys_write+0x42/0x70
 [<c0103f8f>] syscall_call+0x7/0xb

Code: 89 15 00 00 00 00 c3 90 8d 74 26 00 83 ec 0c b8 00 a6
83 d0
```

这个消息是由写入一个由错误模块拥有的设备生成的，这个模块是故意构建用来演示失败的。faulty.c的write方法的实现是非常简单的：

```c
ssize_t faulty_write (struct file *filp, const char __user
*buf, size_t count,

        loff_t *pos)

{


/* 通过解引用一个NULL指针来制造一个简单的错误 */

    *(int *)0 = 0;

    return 0;

}
```

如你所见，我们在这里做的是解引用一个NULL指针。由于0永远不是一个有效的指针值，所以会发生故障，内核将其转化为前面显示的oops消息。然后，调用进程被杀死。

faulty模块在其read实现中有一个不同的故障条件：

```c
ssize_t faulty_read(struct file *filp, char __user *buf,
            size_t count, loff_t *pos)
{
    int ret;
    char stack_buf[4];
    /* 让我们试试缓冲区溢出  */
    memset(stack_buf, 0xff, 20);
    if (count > 4)
        count = 4; /* 复制4个字节到用户 */
    ret = copy_to_user(buf, stack_buf, count);
    if (!ret)
        return count;
    return ret;
}
```

这个方法将一个字符串复制到一个局部变量；不幸的是，字符串比目标数组长。结果的缓冲区溢出在函数返回时导致一个oops。由于返回指令将指令指针带到了无处，这种类型的故障更难追踪，你可能会得到如下的东西：

```shell
EIP: 0010:[<00000000>]
Unable to handle kernel paging request at virtual address
ffffffff

 printing eip:
ffffffff
Oops: 0000 [#5]
SMP
CPU:  0
EIP:  0060:[<ffffffff>]  Not tainted
EFLAGS: 00010296  (2.6.6)
EIP is at 0xffffffff
eax: 0000000c  ebx: ffffffff  ecx: 00000000  edx: bfffda7c
esi: cf434f00  edi: ffffffff  ebp: 00002000  esp: c27fff78
ds: 007b  es: 007b  ss: 0068

Process head (pid: 2331, threadinfo=c27fe000 task=c3226150)
Stack: ffffffff bfffda70 00002000 cf434f20 00000001 00000286
cf434f00 fffffff7 bfffda70 c27fe000 c0150612 cf434f00
bfffda70 00002000 cf434f20 00000000 00000003 00002000
c0103f8f 00000003 bfffda70 00002000 00002000 bfffda70
Call Trace: [<c0150612>] sys_read+0x42/0x70 [<c0103f8f>]
syscall_call+0x7/0xb
Code: Bad EIP value.
```

在这种情况下，我们只看到了调用栈的一部分（vfs_read和faulty_read都缺失），内核抱怨“bad EIP value”。这个抱怨和在开始时列出的冒犯地址（ffffffff）都是内核栈已经被破坏的提示。

通常，当你面对一个oops时，首先要做的是查看问题发生的位置，这通常与调用栈分开列出。在上面显示的第一个oops中，相关的行是： `EIP is at faulty_write+0x4/0x10 [faulty]` 在这里，我们看到我们在faulty_write函数中，该函数位于faulty模块中（在方括号中列出）。十六进制数字表示指令指针在函数的4个字节内，该函数似乎是10（十六进制）字节长。通常，这足以找出问题所在。

如果你需要更多的信息，调用栈会显示你是如何到达事情崩溃的地方的。栈本身以十六进制形式打印；通过一些工作，你通常可以从栈列表中确定局部变量和函数参数的值。有经验的

内核开发人员可以从这里获得一定量的模式识别；例如，如果我们看一下faulty_read oops的栈列表：

```
Stack: ffffffff bfffda70 00002000 cf434f20 00000001 00000286
cf434f00 ffffffff
 bfffda70 c27fe000 c0150612 cf434f00 bfffda70 00002000
cf434f20 00000000
 00000003 00002000 c0103f8f 00000003 bfffda70 00002000
00002000 bfffda70
```

栈顶部的ffffffff是我们破坏事情的字符串的一部分。在x86架构上，默认情况下，用户空间栈开始于0xc0000000以下；因此，重复的值0xbfffda70可能是一个用户空间栈地址；实际上，它是传递给read系统调用的缓冲区的地址，每次它被传递到内核调用链时都会复制。在x86上（再次，默认情况下），内核空间开始于0xc0000000，所以大于这个值的值几乎肯定是内核空间地址，等等。

最后，当查看oops列表时，总是要注意本章开始时讨论的“slab poisoning”值。因此，例如，如果你得到一个内核oops，其中冒犯的地址是0xa5a5a5a5，你几乎肯定忘记在某个地方初始化动态内存。

请注意，只有当你的内核是用CONFIG_KALLSYMS选项打开构建的时候，你才会看到一个符号调用栈（如上所示）。否则，你会看到一个裸的，十六进制的列表，直到你用其他方式解码它，它才有用。

- 当某些比较致命的问题出现时，我们的Linux内核也会抱歉的对我们说：“哎呦(Oops)，对不起，我把事情搞砸了”。Linux内核在发生kernel panic时会打印出Oops信息，把目前的寄存器状态、堆栈内容、以及完整的Call trace都show给我们看，这样就可以帮助我们定位错误。
  linux之Oops原理及解析_linux oops-CSDN博客
  (https://blog.csdn.net/gy794627991/article/details/126650460)

4.5.2. 系统挂起

虽然内核代码中的大多数错误最终都会变成oops消息，但有时它们可能会完全挂起系统。如果系统挂起，不会打印任何消息。例如，如果代码进入一个无限循环，内核停止调度，系统不会对任何操作做出响应，包括魔术Ctrl-Alt-Del组合。你有两种选择来处理系统挂起——要么预先防止它们，要么在事后能够调试它们。

你可以通过在关键点插入调度调用来防止无限循环。调度调用（你可能猜到了）调用调度器，因此，允许其他进程从当前进程中窃取CPU时间。如果一个进程因为你的驱动程序中的

一个错误而在内核空间循环，调度调用使你能够在追踪发生了什么之后杀死进程。

你当然应该意识到，任何对调度的调用都可能创建一个额外的源，用于对你的驱动程序进行重入调用，因为它允许其他进程运行。这种重入性通常不应该是一个问题，假设你在你的驱动程序中使用了适当的锁定。但是，一定要确保在你的驱动程序持有一个自旋锁的任何时候都不要调用调度。

如果你的驱动程序真的挂起了系统，你不知道在哪里插入调度调用，最好的方法可能是添加一些打印消息，并将它们写入控制台（如果需要的话，通过改变console_loglevel值）。

有时候，系统可能看起来已经挂起，但实际上并没有。例如，如果键盘以某种奇怪的方式保持锁定。这些假挂起可以通过查看你为此目的而保持运行的程序的输出来检测。你的显示器上的时钟或系统负载计量器是一个很好的状态监视器；只要它继续更新，调度器就在工作。

对于许多锁定来说，一个不可或缺的工具是“魔术SysRq键”，它在大多数架构上都可用。魔术SysRq是通过PC键盘上的Alt和SysRq键的组合，或者其他平台上的其他特殊键（详见Documentation/sysrq.txt）来调用的，也可以在串行控制台上使用。第三个键，与这两个键一起按下，可以执行一些有用的操作：

按键	功能
r	关闭键盘原始模式；在崩溃的应用程序（如X服务器）可能已经让你的键盘处于奇怪状态的情况下很有用。
k	调用“安全注意键”（SAK）功能。SAK杀死当前控制台上运行的所有进程，留给你一个干净的终端。
s	对所有磁盘进行紧急同步。
u	卸载。试图以只读模式重新挂载所有磁盘。这个操作，通常在s之后立即调用，可以在系统出现严重问题的情况下节省大量的文件系统检查时间。
b	引导。立即重新启动系统。确保先同步和重新挂载磁盘。
p	打印处理器寄存器信息。
t	打印当前任务列表。
m	打印内存信息。

还存在其他的魔术SysRq功能；在内核源代码的Documentation目录的sysrq.txt中查看完整列表。注意，魔术SysRq必须在内核配置中明确启用，出于明显的安全原因，大多数发行版都没有启用它。然而，对于用于开发驱动程序的系统，启用魔术SysRq值得自己构建一个新的内核。魔术SysRq可以通过如下命令在运行时禁用：

```
echo 0 > /proc/sys/kernel/sysrq
```

如果非特权用户可以接触到你的系统键盘，你应该考虑禁用它，以防止意外或故意的损坏。一些早期的内核版本默认禁用了sysrq，所以你需要在运行时通过写入1到同一个/proc/sys文件来启用它。 sysrq操作非常有用，所以它们已经被提供给无法接触控制台的系统管理员。文件/proc/sysrq-trigger是一个只写入口点，你可以通过写入相关的命令字符来触发一个特

定的sysrq操作；然后你可以从内核日志中收集任何输出数据。这个sysrq的入口点总是工作的，即使在控制台上禁用了sysrq。

如果你正在经历一个“活动挂起”，即你的驱动程序陷入循环，但整个系统仍在正常运行，那么有几种技术值得了解。通常，SysRq p功能直接指向了有问题的例程。如果这样做失败，你也可以使用内核的分析功能。构建一个启用了分析的内核，并在命令行上使用profile=2来引导它。使用readprofile工具重置分析计数器，然后让你的驱动程序进入循环。过一会儿，再次使用readprofile来查看内核在哪里花费了时间。另一个更高级的选择是oprofile，你也可以考虑使用。文件Documentation/basic_profiling.txt告诉你开始使用分析器需要知道的一切。

在追踪系统挂起时，值得采取的一个预防措施是将所有的磁盘挂载为只读（或卸载它们）。如果磁盘是只读的或已卸载，就没有损坏文件系统或使其处于不一致状态的风险。另一种可能性是使用一台通过NFS，即网络文件系统，挂载所有文件系统的计算机。必须在内核中启用“NFS-Root”功能，并在引导时传递特殊参数。在这种情况下，你将避免文件系统损坏，甚至不需要使用SysRq，因为文件系统的一致性是由NFS服务器管理的，它不会被你的设备驱动程序关闭。

Linux内核提供了一些与用户空间的通信机制，例如 `procfs` 接口和 `sysfs` 接口，大部分的这些接口都可以作为获取内核信息的手段。

但除了这些接口，内核也提供了专门的调试机制——系统请求键 `SysRq` 。

`SysRq` 被内核称为“`Magic SysRq key`”，即“**神奇的系统请求键**”。

简单来说，就是可以**通过键盘的按键获取内核的信息**，用于调试。相当于是一个快捷键。

要使用系统请求键 `SysRq` ，内核配置选项中必须打开 `CONFIG_MAGIC_SYSRQ`

```c
CONFIG_MAGIC_SYSRQ=y
```

`SysRq` 键是复合键【 `Alt` + `SysRq` 】,大多数键盘的 `SysRq` 和 `PrtSc` 键是复用的。

按住 `SysRq` 复合键， 再输入第三个命令键， 可以执行相应的系统调试命令。 例如，输入 `t` 键， 可以得到当前运行的进程和所有进程的堆栈跟踪。 回溯跟踪将被写到 `/var/log/messages` 文件中。 如果内核都配置好了， 系统应该已经转换了内核的符号地址。

知 **Linux内核基础篇——神奇的系统请求键SysRq - 知乎 (zhihu.com)**
(https://zhuanlan.zhihu.com/p/641038376)

4.6. 调试器和相关工具

在调试模块时的最后手段是使用调试器逐步执行代码，观察变量和机器寄存器的值。这种方法耗时且应尽可能避免。然而，通过调试器实现的对代码的细粒度视角有时是无价的。

在内核上使用交互式调试器是一项挑战。内核代表系统上的所有进程在其自己的地址空间中运行。因此，用户空间调试器提供的许多常见功能，如断点和单步执行，在内核中更难获得。在这一节中，我们将看几种调试内核的方法；每种方法都有其优点和缺点。

4.6.1. 使用 gdb

gdb对于查看系统内部非常有用。在这个级别熟练使用调试器需要对gdb命令有一定的熟悉度，对目标平台的汇编代码有一些理解，以及能够匹配源代码和优化后的汇编。

必须像调用应用程序一样调用调试器。除了指定ELF内核映像的文件名外，你还需要在命令行上提供核心文件的名称。对于正在运行的内核，该核心文件是内核核心映像，/proc/kcore。gdb的典型调用如下所示：

```
gdb /usr/src/linux/vmlinux /proc/kcore
```

第一个参数是未压缩的ELF内核可执行文件的名称，而不是zImage或bzImage或任何专门为引导环境构建的东西。

gdb命令行上的第二个参数是核心文件的名称。像/proc中的任何文件一样，/proc/kcore在读取时生成。当在/proc文件系统中执行读取系统调用时，它映射到一个数据生成函数，而不是数据检索函数；我们已经在本章前面的“使用/proc文件系统”一节中利用了这个特性。kcore用于以核心文件的格式表示内核“可执行文件”；它是一个巨大的文件，因为它代表了整个内核地址空间，对应于所有物理内存。在gdb内部，你可以通过发出标准的gdb命令来查看内核变量。例如，`p jiffies`打印从系统引导到当前时间的时钟滴答数。

- 内核可以通过proc文件系统，将当前正在运行内核的core文件通过/proc/kcore导出，gdb可通过这个core文件实现对内核的在线调试。由于gdb本身运行在linux的用户态中，因此这种方法只能执行一些非侵入式的调试操作，如查看变量，寄存器等，但不能对内核执行像打断点、单步调试以及修改变量之类的调试命令。
- 知 linux内核调试（六）如何用gdb调试内核 - 知乎 (zhihu.com) (https://zhuanlan.zhihu.com/p/546089584)

当你从gdb打印数据时，内核仍在运行，各种数据项在不同的时间有不同的值；然而，gdb通过缓存已经读取的数据来优化对核心文件的访问。如果你试图再次查看jiffies变量，你会得到和之前相同的答案。为了避免额外的磁盘访问而缓存值是对传统核心文件的正确行为，但是当使用“动态”核心映像时，这是不方便的。解决方案是每当你想刷新gdb缓存时，发出命令

`core-file /proc/kcore`；调试器准备使用一个新的核心文件，并丢弃任何旧的信息。然而，你不总是需要在读取新的数据时发出core-file命令；gdb以几千字节的块读取核心，并只缓存它已经引用的块。

通常由gdb提供的许多功能在你使用内核时是不可用的。例如，gdb不能修改内核数据；它期望在玩弄其内存映像之前运行一个要在其自己的控制下调试的程序。也不可能设置断点或监视点，或者通过内核函数进行单步执行。

请注意，为了让gdb有符号信息可用，你必须将你的内核编译为设置了`CONFIG_DEBUG_INFO`选项。结果是磁盘上的内核映像大得多，但是，没有这些信息，几乎不可能深入内核变量。

有了调试信息，你可以了解内核内部发生了什么。gdb愉快地打印出结构，跟踪指针等。然而，更难的一件事是检查模块。由于模块不是传递给gdb的vmlinux映像的一部分，调试器对它们一无所知。幸运的是，从内核2.6.7开始，可以教gdb知道它需要知道的东西来检查可加载的模块。

Linux可加载模块是ELF格式的可执行映像；因此，它们已被划分为许多部分。一个典型的模块可以包含十几个或更多的部分，但在调试会话中通常有三个是相关的：
.text 这个部分包含模块的可执行代码。调试器必须知道这个部分在哪里，才能给出回溯或设置断点。（当在/proc/kcore上运行调试器时，这两个操作都不相关，但是当使用kgdb时，它们可能很有用，下面会描述）。
.bss .data 这两个部分保存了模块的变量。任何在编译时没有初始化的变量都会在.bss中，而那些已经初始化的变量会进入.data。

让gdb与可加载模块一起工作需要告诉调试器给定模块的部分已经加载到哪里。这个信息可以在sysfs中找到，位于/sys/module下。例如，在加载scull模块后，目录/sys/module/scull/sections包含了如.text这样的文件名；每个文件的内容是该部分的基地址。

现在我们可以发出一个gdb命令来告诉它我们的模块了。我们需要的命令是`add-symbol-file`；这个命令接受模块对象文件的名称，.text基地址，以及一系列描述已经放置了任何其他感兴趣的部分的可选参数。在通过sysfs中的模块部分数据后，我们可以构造一个命令，如下所示：

```shell
(gdb) add-symbol-file ../scull.ko 0xd0832000 \

-s .bss 0xd0837100 \

-s .data 0xd0836be0
```

我们在示例源码中包含了一个小脚本（gdbline），它可以为给定的模块创建这个命令。

现在我们可以使用gdb来检查我们可加载模块中的变量。这里有一个从scull调试会话中取得的快速示例：

```shell
(gdb) add-symbol-file scull.ko 0xd0832000 \

-s .bss 0xd0837100 \

-s .data 0xd0836be0

add symbol table from file "scull.ko" at

.text_addr = 0xd0832000

.bss_addr = 0xd0837100

.data_addr = 0xd0836be0

(y or n) y

Reading symbols from scull.ko...done.

(gdb) p scull_devices[0]

$1 = {data = 0xcfd66c50,

quantum = 4000,

qset = 1000,

size = 20881,

access_key = 0,

...}
```

这里我们看到第一个scull设备当前持有20881字节。如果我们愿意，我们可以跟踪数据链，或者查看模块中的其他任何感兴趣的东西。

还有一个值得知道的有用的技巧是这个：

```shell
(gdb) print *(address)
```

在这里，为address填入一个十六进制地址；输出是对应于该地址的代码的文件和行号。例如，这种技术可能用于找出函数指针真正指向哪里。

我们仍然不能执行像设置断点或修改数据这样的典型调试任务；要执行这些操作，我们需要使用像kdb（接下来描述）或kgdb（我们马上就要讲到）这样的工具。
gdb调试笔记_add-symbol-file-CSDN博客
(https://blog.csdn.net/faxiang1230/article/details/108848470)

4.6.2. kdb 内核调试器

许多读者可能会想知道为什么内核没有内置更多的高级调试功能。答案很简单，那就是Linus不相信交互式调试器。他担心它们会导致糟糕的修复，这些修复只是修补症状，而不是解决问题的真正原因。因此，没有内置的调试器。

然而，其他内核开发者偶尔会使用交互式调试工具。一个这样的工具是kdb内置内核调试器，可以从oss.sgi.com获取非官方的补丁。要使用kdb，你必须获取补丁（确保获取一个与你的内核版本匹配的版本），应用它，然后重新构建和安装内核。请注意，截至撰写本文时，kdb只在IA-32（x86）系统上工作（尽管在主线内核源码中有一段时间存在IA-64版本，但后来被移除）。

一旦你运行了一个启用了kdb的内核，有几种方法可以进入调试器。在控制台上按暂停（或中断）键会启动调试器。当内核oops发生或者当断点被触发时，kdb也会启动。无论哪种情况，你会看到类似这样的消息：

```shell
Entering kdb (0xc0347b80) on processor 0 due to Keyboard
Entry

[0]kdb>
```

请注意，当kdb运行时，内核做的几乎所有事情都会停止。在你调用kdb的系统上不应该运行其他任何东西；特别是，你不应该打开网络——除非，当然，你正在调试一个网络驱动程序。如果你将使用kdb，通常最好在单用户模式下启动系统。

作为一个例子，考虑一个快速的scull调试会话。假设驱动程序已经加载，我们可以告诉kdb在scull_read中设置一个断点，如下所示：

```shell
[0]kdb> bp scull_read

Instruction(i) BP #0 at 0xcd087c5dc (scull_read)

is enabled globally adjust 1

[0]kdb> go
```

bp命令告诉kdb下次内核进入scull_read时停止。然后你输入go继续执行。在将一些东西放入一个scull设备后，我们可以尝试通过在另一个终端上的shell下运行cat来读取它，得到以下结果：

```shell
Instruction(i) breakpoint #0 at 0xd087c5dc (adjusted)

0xd087c5dc scull_read:          int3

Entering kdb (current=0xcf09f890, pid 1575) on processor 0
due to

Breakpoint @ 0xd087c5dc

[0]kdb>
```

现在我们位于scull_read的开始处。要看我们是如何到达那里的，我们可以获取一个堆栈跟踪：

```
[0]kdb> bt
ESP    EIP        Function (args)
0xcdbddf74 0xd087c5dc [scull]scull_read
0xcdbddf78 0xc0150718 vfs_read+0xb8
0xcdbddfa4 0xc01509c2 sys_read+0x42
0xcdbddfc4 0xc0103fcf syscall_call+0x7
[0]kdb>
```

kdb试图打印出调用跟踪中每个函数的参数。然而，它被编译器使用的优化技巧弄糊涂了。因此，它未能打印出scull_read的参数。
知 linux内核调试（七）使用kdb/kgdb调试内核 - 知乎 (zhihu.com)
(https://zhuanlan.zhihu.com/p/546416941)

4.6.3. kgdb 补丁

到目前为止，我们看到的两种交互式调试方法（在/proc/kcore上使用gdb和kdb）都无法达到用户空间应用程序开发人员习惯的环境。如果有一个真正的内核调试器，支持改变变量、断点等功能，那不是很好吗？

事实证明，这样的解决方案确实存在。在撰写本文时，有两个单独的补丁在流通，它们允许gdb以完全的功能运行在内核上。令人困惑的是，这两个补丁都被称为kgdb。它们通过分离运行测试内核的系统和运行调试器的系统来工作；这两者通常通过串行电缆连接。因此，开发者可以在他或她稳定的桌面系统上运行gdb，同时操作运行在一个可牺牲的测试盒子上的内核。在开始时设置gdb的这种模式需要一些时间，但是当一个困难的bug出现时，这种投资可以迅速得到回报。

这些补丁处于强烈的变动状态，甚至可能在某一点合并，所以我们避免对它们说太多，除了它们在哪里和它们的基本特性。感兴趣的读者被鼓励去看看当前的情况。

第一个kgdb补丁目前在-mm内核树中找到——这是一个补丁进入2.6主线的暂存区。这个版本的补丁支持x86、SuperH、ia64、x86_64、SPARC和32位PPC架构。除了通常通过串行端口操作的模式，这个版本的kgdb还可以通过局域网进行通信。只需启用以太网模式，并以

kgdboe参数启动，以指示可以发出调试命令的IP地址。Documentation/i386/kgdb下的文档描述了如何设置。

作为替代，你可以使用在 **http://kgdb.sf.net/上找到的kgdb补丁。这个版本的调试器不支持网络通信模式（虽然据说正在开发中），但它确实有一些内置的支持，用于处理可加载模块。它支持x86、x86_64、PowerPC和S/390架构。(http://kgdb.sf.net/%E4%B8%8A%E6%89%BE%E5%88%B0%E7%9A%84kgdb%E8%A1%A5%E4%B8%81%E3%80%82%E8%BF%99%E4%B8%AA%E7%89%88%E6%9C%AC%E7%9A%84%E8%B0%83%E8%AF%95%E5%99%A8%E4%B8%8D%E6%94%AF%E6%8C%81%E7%BD%91%E7%BB%9C%E9%80%9A%E4%BF%A1%E6%A8%A1%E5%BC%8F%EF%BC%88%E8%99%BD%E7%84%B6%E6%8D%AE%E8%AF%B4%E6%AD%A3%E5%9C%A8%E5%BC%80%E5%8F%91%E4%B8%AD%EF%BC%89%EF%BC%8C%E4%BD%86%E5%AE%83%E7%A1%AE%E5%AE%9E%E6%9C%89%E4%B8%80%E4%BA%9B%E5%86%85%E7%BD%AE%E7%9A%84%E6%94%AF%E6%8C%81%EF%BC%8C%E7%94%A8%E4%BA%8E%E5%A4%84%E7%90%86%E5%8F%AF%E5%8A%A0%E8%BD%BD%E6%A8%A1%E5%9D%97%E3%80%82%E5%AE%83%E6%94%AF%E6%8C%81x86%E3%80%81x86_64%E3%80%81PowerPC%E5%92%8CS/390%E6%9E%B6%E6%9E%84%E3%80%82)**

4.6.4. 用户模式 Linux 移植

用户模式Linux（UML）是一个有趣的概念。它被构造为Linux内核的一个单独端口，有自己的arch/um子目录。然而，它并不运行在新类型的硬件上；相反，它在Linux系统调用接口上实现的虚拟机上运行。因此，UML允许Linux内核作为一个单独的用户模式进程在Linux系统上运行。

有一个内核副本作为用户模式进程运行带来了许多优点。因为它在一个受限的虚拟处理器上运行，一个有bug的内核不能破坏“真实”的系统。可以在同一台机器上轻松尝试不同的硬件和软件配置。而且，对于内核开发者来说，可能最重要的是，用户模式内核可以很容易地用gdb或其他调试器进行操作。毕竟，它只是另一个进程。UML显然有加速内核开发的潜力。

然而，从驱动程序编写者的角度看，UML有一个很大的缺点：用户模式内核无法访问主机系统的硬件。因此，虽然它可以用于调试本书中的大多数示例驱动程序，但UML还不能用于调试必须处理真实硬件的驱动程序.

4.6.5. Linux 追踪工具

Linux跟踪工具包（LTT）是一个内核补丁和一组相关的实用程序，它们允许跟踪内核中的事件。跟踪包括时间信息，可以创建一个相当完整的图片，显示在给定时间段内发生了什么。因此，它不仅可以用于调试，还可以用于追踪性能问题。

LTT的工作方式是在内核中插入一系列的跟踪点。每当内核通过这些跟踪点时，就会生成一个事件，该事件被记录在一个特殊的跟踪缓冲区中。然后，可以使用一系列的用户空间工具来查看和分析这些跟踪数据。

LTT的一个主要优点是它对系统的影响非常小。这使得它可以在生产系统上使用，而不会对系统性能产生太大影响。然而，LTT的一个缺点是它需要修改内核源代码以插入跟踪点。这意味着你需要重新编译和安装你的内核，这可能不是所有人都愿意做的。

4.6.6. 动态探针

动态探针（或DProbes）是IBM为IA-32架构的Linux发布的一个调试工具（在GPL下）。它允许在系统的几乎任何地方（包括用户空间和内核空间）放置一个“探针”。探针由一些代码（用一种专门的、基于堆栈的语言编写）组成，当控制命中给定点时执行。这段代码可以将信息报告回用户空间，改变寄存器，或做其他一些事情。DProbes的有用特性是，一旦将能力构建到内核中，就可以在运行的系统中的任何地方插入探针，而无需内核构建或重启。DProbes还可以与LTT一起工作，在任意位置插入新的跟踪事件。

DProbes的一个主要优点是它可以在运行时动态地插入和删除探针，而无需重新编译或重启内核。这使得它非常适合于生产环境，因为你可以在不中断服务的情况下进行调试。

然而，DProbes的一个缺点是它需要一些特殊的知识来编写探针代码。这可能会对一些没有这方面经验的开发者构成挑战。