Foreword

作者声明

《Linux 内核驱动模块编程指南》最初是由 Ori Pomerantz 为 2.2 版本的内核编写的 ，后来，Ori 将文档维护的任务交给了 Peter Jay Salzman，Peter 完成了 2.4 内核版本文档 的编写，毕竟 Linux 内核驱动模块是一个更新很快的内容。现在，Peter 也无法腾出足够的 时间来完成 2.6 内核版本文档的编写，目前该 2.6 内核版本的文档由合作者 Michael Burian 完成。

版本和注意

Linux 内核模块是一块不断更新进步的内容，在 LKMPG 上总有关于是否保留还是历史 版本的争论。Michael 和我最终是决定为每个新的稳定版本内核建立一个新的文档分支。也 就是说 LKMPG 2.4.x 专注于 2.4 的内核，而 LKMPG 2.6.x 将专注于 2.6 的内核。我们不会在一 篇文档中提供对旧版本内核的支持，对此感兴趣的读者应该寻找相关版本的文档分支。

在文档中的绝大部分源代码和讨论都应该适用于其它平台，但我无法提供任何保证。其中的一个例外就是 Chapter 12, 中断处理该章的源代码和讨论就只适用于 x86 平台。

感谢

感谢下列人士为此文档提供了他们宝贵的意见。他们是：Ignacio Martin, David Porter, Daniele Paolo,Scarpazza 和 Dimo Velev。

译者注

我，译者，名叫田竞，目前是一名在北京邮电大学就读的通信专业的大学生。 自高中起我就是 Linux 的爱好者并追随至今。我喜欢 Linux 给我带来的自由，我想大家也一样。没有人不向往自由。

我学习内核模块编写时最初阅读的是 Orelly 出版社的使用 2.0 版本的内核的书籍,但如同我预料的一样，书中的许多事例由于内核代码的变化而无法使用。这让想亲自体验内核模块的神秘的我非常苦恼。 我在 Linux 文档计划在上海的镜像站 ldp.linuxforum.net 上找到了这篇文档。

受原作者 Ori 的鼓励，基于上次完成的 LKMPG 2.4 的，内容有稍许的改变和扩充。应该是目前最新的了。 翻译的方式有所改变，在基于 LDP 认可的 docbook 格式上翻译，通过 docbook2html 转换为附件中的 html 文档。 由于对 docbook 不是很熟悉，其中的一些标题尚未翻译，而且可能破坏了原有的 tag，导致 html 出现一些错误显示， 但总体来说很少。修改了很多 2.4 中的错别字。
学习并分享学习的过程是我翻译的最终目的。

Chapter 1. Introduction

什么是内核模块？

现在，你是不是想编写内核模块。你应该懂得 C 语言，写过一些用户程序， 那么现在你将要见识一些真实的东西。在这里，你会看到一个野蛮的指针是如何 毁掉你的文件系统的，一次内核崩溃意味着重启动。

什么是内核模块？内核模块是一些可以让操作系统内核在需要时载入和执 行的代码，这同样意味着它可以在不需要时有操作系统卸载。它们扩展了操作系 统内核的功能却不需要重新启动系统。举例子来说，其中一种内核模块时设备驱 动程序模块，它们用来让操作系统正确识别，使用安装在系统上的硬件设备。如 果没有内核模块，我们不得不一次又一次重新编译生成单内核操作系统的内核镜 像来加入新的功能。这还意味着一个臃肿的内核。

内核模块是如何被调入内核工作的？

你可以通过执行 lsmod 命令来查看内核已经加载了哪 些内核模块，该命令通过读取/proc/modules 文件的内容 来获得所需信息。

这些内核模块是如何被调入内核的？当操作系统内核需要的扩展功能不存 在时，内核模块管理守护进程 kmod[1]执行 modprobe 去加载内核模 块。两种类型的参数被传递给 modprobe：

一个内核模块的名字像 softdog 或是 ppp。

通用识别符像 char-major-10-30。

当传递给 modprobe 是通用识别符时，modprobe 首先在文件 /etc/modules.conf 查找该字符串。如果它发现的一行别名像：

alias char-major-10-30 softdog

它就明白通用识别符是指向内核模块 softdog.o。

然后，modprobe 遍历文件/lib/modules/version/modules.dep 来判断是否有其它内核模块需要在该模块加载前被加载。该文件是由命令 depmod -a 建立，保存着内核模块的依赖关系。举例来说，msdos.o 依赖于模块 fat.o 内核模块已经被内核载入。当要加载的内核模块需要使用别的模块提供的符号链接时（多半是变量或函数）， 那么那些提供这些所需符号链接的内核模块就被该模块所依赖。

最终，modprobe 调用 insmod 先加载被依赖的模块，然后加载该被内核要求的模块。modprobe 将 insmod 指向 /lib/modules/version/[2]目录，该目录为默认标准存放内核模块的目录。insmod 对内核模块存放位置 的处理相当呆板，所以 modprobe 应该很清楚的知道默认标准的内核模块存放的位置。所以，当你想要载入一个内 核模块时，你可以执行：

```
insmod /lib/modules/2.5.1/kernel/fs/fat/fat.o
insmod /lib/modules/2.5.1/kernel/fs/msdos/msdos.o
```

或只是执行"modprobe -a msdos"。

Linux 提供 modprobe, insmod and depmod 在一个名为 modutils 或 mod-utils 的工具包内。

在结束本章前，让我们来看一个 /etc/modules.conf 文件：

```
#This file is automatically generated by update-modules
path[misc]=/lib/modules/2.4.?/local
keep
path[net]=~p/mymodules
options mydriver irq=10
alias eth0 eepro
```

用'#'起始的行为注释。空白行被忽略。

以 path[misc]起始的行告诉 modprobe 用 /lib/modules/2.4.?/local 替代搜寻 misc 内核模块的路径。正如你看到的，命令解释器 shell 的元字符也可以使用。

以 path[net]起始的行告诉 modprobe 在目录 ~p/mymodules 搜索网络方面的内核模块。但是，在 path[net] 指令之前使用的"keep" 指令告诉 modprobe 只是将该路径添加到标准搜索路径中，而不是像对待 misc 前面那样进行替换。

以 alias 起始的的行使 modprobe 加载 eepro.o 当 kmod 以通用识别符'eth0' 要求加载相应内核模块时。

你不会发现像 "alias block-major-2 floppy" 这样的别名行在文件 /etc/modules.conf 因为 modprobe 已经知道在绝大多数系统上安装的标准的设备的驱动模块。

现在你已经知道内核模块是如何被调入的了。当你想写你自己的依赖于其它模块的内核模块时， 还有一些内容没有提供。这个相对高级的问题将在以后的章节中介绍，当我们已经完成前面的学习后。

在开始前

在我们介绍源代码前，有一些事需要注意。系统彼此之间的不同会导致许多困难。 顺利的编译并且加载你的第一个"hello world"模块有时就会比较困难。但是当你跨过 这道坎时，后面会顺利的多。

内核模块和内核的版本问题

为某个版本编译的模块将不能被另一个版本的内核加载如果内核中打开了 CONFIG_MODVERSIONS 选项。我们暂时不会讨论与此相关的 内容。在我们进入相关内容前，本文档中的范例可能在该选项打开的情况下无法 工作。但是，目前绝大多数的发行版是将该选项打开的。所以如果你遇到和版本 相关的错误时，最好，重新编译一个关闭该选项的内核。

使用 X 带来的问题

强烈建议你在控制台下输入文档中的范例代码，编译然后加载模块，而不是在 X 下。

模块不能像 printf()那样输出到屏幕，但它们可以 记录信息和警告，当且仅当你在使用控制台时这些信息才能最终显示在屏幕上。 如果你从 xterm 中 insmod 一个模块，这些日志信息只会记录在你的日志文件中。 除了查看日志文件你将无法 得到输出信息。想要及时的获得这些日志信息，建议 所有的工作都在控制台下进行。

编译相关和内核版本相关的问题
Linux 的发行版经常给内核打一些非标准的补丁，这种情况回导致一些问题的发生。

一个更普遍的问题是一些 Linux 发行版提供的头文件不完整。编译模块时你将需要非常多 的内核头文件。墨菲法则之一就是那些缺少的头文件恰恰是你最需要的。

我强烈建议从 Linux 镜像站点下载源代码包，编译新内核并用新内核启动系统来避免以上 的问题。参阅 "Linux Kernel HOWTO"获得详细内容。

具有讽刺意味的是，这也会导致一些问题。gcc 倾向于在缺省的内核源文件路径(通常是/usr/src/)下寻找源代码文件。这可以通过 gcc 的-I 选项来切换。

Notes
[1] 在早期的 linux 版本中，是 一个名为 kerneld 的守护进程。

[2] 如果你在修改内核，为避免 覆盖你现在工作的模块，你应该试试使用内核 Makefile 中的变量 EXTRAVERSION 去建立一个独 立的模块目录。

Chapter 2. Hello World

Hello, World (part 1): 最简单的内核模块

当第一个洞穴程序员在第一台洞穴计算机的墙上上凿写第一个程序时， 这是一个在羚羊皮上输出`Hello, world'的字符串。罗马的编程书籍上是以 `Salut, Mundi'这样的程序开始的。 我不明白人们为什么要破坏这个传统， 但我认为还是不明白为好。我们将从编写一系列的`Hello, world'模块开始， 一步步展示编写内核模块的基础的方方面面。

这可能是一个最简单的模块了。先别急着编译它。我们将在下章模块编译的章节介绍相关内容。

Example 2-1. hello-1.c

```
/*
*  hello-1.c - The simplest kernel module.
*/
#include <linux/module.h>         /* Needed by all modules */
#include <linux/kernel.h>         /* Needed for KERN_ALERT */

int init_module(void)
{
        printk("<1>Hello world 1.\n" );

        /*
         * A non 0 return means init_module failed; module can't be loaded.
         */
        return 0;
}

void cleanup_module(void)
{
        printk(KERN_ALERT "Goodbye world 1.\n" );
}
```

一个内核模块应该至少包含两个函数。一个"开始"(初始化)的函数被称为 init_module() 还有一个"结束"(干一些收尾清理的工作)的函数被称为cleanup_module() ，当内核模块被rmmod 卸载时被执行。实际上，从内核版本 2.3.13 开始这种情况有些改变。 你可以为你的开始和结束函数起任意的名字。 你将在以后学习如何实现这一点 the Section called Hello World (part 2)。 实际上，这个新方法时推荐的实现方法。但是，许多人仍然使 init_module()和 cleanup_module()作为他们的开始和结束函数。

一般，init_module()要么向内核注册它可以处理的事物，要么用自己的代码 替代某个内核函数(代码通常这样做然后再去调用原先的函数代码)。函数 cleanup_module()应该撤消任何 init_module()做的事，从而 内核模块可以被安全的卸载。

最后，任一个内核模块需要包含 linux/module.h。 我们仅仅需要包含 linux/kernel.h 当需要使用 printk()记录级别的宏扩展时 KERN_ALERT，相关内容将在 the Section called 介绍 printk()中介绍。

介绍 printk()

不管你可能怎么想，printk()并不是设计用来同用户交互的，虽然我们在 hello-1 就是出于这样的目的使用它！它实际上是为内核提供日志功能， 记录内核信息或用来给出警告。因此，每个 printk() 声明都会带一个优先级，就像你看到的<1>和 KERN_ALERT 那样。内核总共定义了八个优先级的宏， 所以你不必使用晦涩的数字代码，并且你可以从文件 linux/kernel.h 查看这些宏和它们的意义。如果你 不指明优先级，默认的优先级 DEFAULT_MESSAGE_LOGLEVEL 将被采用。

阅读一下这些优先级的宏。头文件同时也描述了每个优先级的意义。在实际中， 使用宏而不要使用数字，就像<4>。总是使用宏，就像 KERN_WARNING。

当优先级低于 int console_loglevel，信息将直接打印在你的终端上。如果同时 syslogd 和 klogd 都在运行，信息也同时添加在文件 /var/log/messages，而不管是否显示在控制台上与否。我们使用像 KERN_ALERT 这样的高优先级，来确保 printk()将信息输出到 控制台而不是只是添加到日志文件中。当你编写真正的实用的模块时，你应该针对可能遇到的情况使用合 适的优先级。

编译内核模块

内核模块在用 gcc 编译时需要使用特定的参数。另外，一些宏同样需要定义。 这是因为在编译成可执行文件和内核模块时， 内核头文件起的作用是不同的。 以往的内核版本需要我们去在 Makefile 中手动设置这些设定。尽管这些 Makefile 是按目录分层次 安排的，但是这其中有许多多余的重复并导致代码树大而难以维护。 幸运的是，一种称为 kbuild 的新方法被引入，现在外部的可加载内核模块的编译的方法已经同内核编译统一起来。想了解更多的编 译非内核代码树中的模块(就像我们将要编写的)请参考帮助文件 linux/Documentation/kbuild/modules.txt。

现在让我们看一个编译名字叫做 hello-1.c 的模块的简单的 Makefile 文件：

Example 2-2. 一个基本的 Makefile

```
obj-m += hello-1.o
```

现在你可以通过执行命令 make -C /usr/src/linux-`uname -r` SUBDIRS=$PWD modules 编译模块。 你应该得到同下面类似的屏幕输出：

```
[root@pcsenonsrv test_module]# make -C /usr/src/linux-`uname -r` SUBDIRS=$PWD modules
make: Entering directory `/usr/src/linux-2.6.x
  CC [M]  /root/test_module/hello-1.o
  Building modules, stage 2.
  MODPOST
  CC      /root/test_module/hello-1.mod.o
  LD [M]  /root/test_module/hello-1.ko
make: Leaving directory `/usr/src/linux-2.6.x
```

请注意 2.6 的内核现在引入一种新的内核模块命名规范：内核模块现在使用.ko 的文件后缀(代替 以往的.o 后缀)，这样内核模块就可以同普通的目标文件区别开。更详细的文档请参考 linux/Documentation/kbuild/makefiles.txt。在研究 Makefile 之前请确认你已经参考了这些文档。

现在是使用 insmod ./hello-1.ko 命令加载该模块的时候了(忽略任何你看到的关于内核污染的输出 显示，我们将在以后介绍相关内容)。

所有已经被加载的内核模块都罗列在文件/proc/modules 中。cat 一下这个文件看一下你的模块是否真的 成为内核的一部分了。如果是，祝贺你！你现在已经是内核模块的作者了。当你的新鲜劲过去后，使用命令 rmmod hello-1.卸载模块。再看一下/var/log/messages 文件的内容是否有相关的日志内容。

这儿是另一个练习。看到了在声明 init_module()上的注释吗? 改变返回值非零，重新编译再加载，发生了什么？

Hello World (part 2)

在内核 Linux 2.4 中，你可以为你的模块的"开始"和"结束"函数起任意的名字。它们不再必须使用 init_module()和 cleanup_module()的名字。这可以通过宏 module_init()和 module_exit()实现。这些宏在头文件 linux/init.h 定义。唯一需要注意的地方是函数必须在宏的使用前定义,否则会有编译 错误。下面就是一个例子。

Example 2-3. hello-2.c

```
/*
*  hello-2.c - Demonstrating the module_init() and module_exit() macros.
*  This is preferred over using init_module() and cleanup_module().
*/
#include <linux/module.h>         /* Needed by all modules */
#include <linux/kernel.h>         /* Needed for KERN_ALERT */
#include <linux/init.h>                 /* Needed for the macros */

static int __init hello_2_init(void)
{
        printk(KERN_ALERT "Hello, world 2\n" );
        return 0;
}

static void __exit hello_2_exit(void)
{
        printk(KERN_ALERT "Goodbye, world 2\n" );
}

module_init(hello_2_init);
module_exit(hello_2_exit);
```

现在我们已经写过两个真正的模块了。添加编译另一个模块的选项十分简单，如下：

Example 2-4. 两个内核模块使用的 Makefile

```
obj-m += hello-1.o
obj-m += hello-2.o
```

现在让我们来研究一下 linux/drivers/char/Makefile 这个实际中的例子。就如同你看到的， 一些被编译进内核 (obj-y),但是这些 obj-m 哪里去了呢？对于熟悉 shell 脚本的人这不难理解。这些在 Makefile 中随处可见 的 obj-$(CONFIG_FOO)的指令将会在 CONFIG_FOO 被设置后扩展为你熟悉的 obj-y 或 obj-m。这其实就是你在使用 make menuconfig 编译内核时生成的 linux/.config 中设置的东西。

Hello World (part 3): 关于__init 和__exit 宏

这里展示了内核 2.2 以后引入的一个新特性。注意在负责"初始化"和"清理收尾"的函数定义处的变化。宏 __init 的使用会在初始化完成后丢弃该函数并收回所占内存，如果该模块被编译进内核，而不是动态加载。

宏__initdata 同__init 类似，只不过对变量有效。

宏__exit 将忽略"清理收尾"的函数如果该模块被编译进内核。同宏 __exit 一样，对动态加载模块是无效的。这很容易理解。编译进内核的模块 是没有清理收尾工作的，而动态加载的却需要自己完成这些工作。

这些宏在头文件 linux/init.h 定义，用来释放内核占用的内存。 当你在启动时看到这样的 Freeing unused kernel memory: 236k freed 内核输出，上面的 那些正是内核所释放的。

Example 2-5. hello-3.c

```
/*
*  hello-3.c - Illustrating the __init, __initdata and __exit macros.
*/
#include <linux/module.h>         /* Needed by all modules */
#include <linux/kernel.h>         /* Needed for KERN_ALERT */
#include <linux/init.h>                 /* Needed for the macros */

static int hello3_data __initdata = 3;

static int __init hello_3_init(void)
{
        printk(KERN_ALERT "Hello, world %d\n", hello3_data);
        return 0;
}

static void __exit hello_3_exit(void)
{
        printk(KERN_ALERT "Goodbye, world 3\n" );
}

module_init(hello_3_init);
module_exit(hello_3_exit);
```

Hello World (part 4): 内核模块证书和内核模块文档说明

如果你在使用 2.4 或更新的内核，当你加载你的模块时，你也许注意到了这些输出信息：

```
# insmod hello-3.o
Warning: loading hello-3.o will taint the kernel: no license
  See http://www.tux.org/lkml/#export-tainted for information about tainted modules
Hello, world 3
Module hello-3 loaded, with warnings
```

在 2.4 或更新的内核中，一种识别代码是否在 GPL 许可下发布的机制被引入， 因此人们可以在使用非公开的源代码产品时得到警告。这通过在下一章展示的宏 MODULE_LICENSE()当你设置在 GPL 证书下发布你的代码时， 你可以取消这些警告。这种证书机制在头文件 linux/module.h 实现，同时还有一些相关文档信息。

```
/*
* The following license idents are currently accepted as indicating free
* software modules
*
*       "GPL "                          [GNU Public License v2 or later]
*       "GPL v2 "                       [GNU Public License v2]
*       "GPL and additional rights"     [GNU Public License v2 rights and more]
*       "Dual BSD/GPL "                 [GNU Public License v2
*                                      or BSD license choice]
*       "Dual MPL/GPL "                 [GNU Public License v2
*                                      or Mozilla license choice]
*
* The following other idents are available
*
*       " Proprietary"                  [Non free products]
*
* There are dual licensed components, but when running with Linux it is the
* GPL that is relevant so this is a non issue. Similarly LGPL linked with GPL
* is a GPL combined work.
*
* This exists for several reasons
* 1.        So modinfo can show license info for users wanting to vet their setup
*       is free
* 2.        So the community can ignore bug reports including proprietary modules
* 3.        So vendors can do likewise based on their own policies
*/
```

类似的，宏 MODULE_DESCRIPTION()用来描述模块的用途。 宏 MODULE_AUTHOR()用来声明模块的作者。宏 MODULE_SUPPORTED_DEVICE() 声明模块支持的设备。

这些宏都在头文件 linux/module.h 定义，并且内核本身并不使用这些宏。它们只是用来提供识别信息，可用工具程序像 objdump 查看。 作为一个练习，使用 grep 从目录 linux/drivers 看一看这些模块的作者是如何 为他们的模块提供识别信息和档案的。

Example 2-6. hello-4.c

```
/*
*  hello-4.c - Demonstrates module documentation.
*/
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#define DRIVER_AUTHOR " Peter Jay Salzman <p@dirac.org>"
#define DRIVER_DESC   "A sample driver"

static int __init init_hello_4(void)
{
        printk(KERN_ALERT "Hello, world 4\n" );
        return 0;
}

static void __exit cleanup_hello_4(void)
{
        printk(KERN_ALERT "Goodbye, world 4\n" );
}

module_init(init_hello_4);
module_exit(cleanup_hello_4);

/*
*  You can use strings, like this:
*/

/*
* Get rid of taint message by declaring code as GPL.
*/
MODULE_LICENSE("GPL" );

/*
* Or with defines, like this:
*/
MODULE_AUTHOR(DRIVER_AUTHOR );          /* Who wrote this module? */
MODULE_DESCRIPTION(DRIVER_DESC );          /* What does this module do */

/*
*  This module uses /dev/testdevice.  The MODULE_SUPPORTED_DEVICE macro might
*  be used in the future to help automatic configuration of modules, but is
*  currently unused other than for documentation purposes.
*/
MODULE_SUPPORTED_DEVICE("testdevice" );
```

从命令行传递参数给内核模块

模块也可以从命令行获取参数。但不是通过以前你习惯的 argc/argv。

要传递参数给模块，首先将获取参数值的变量声明为全局变量。然后使用宏 MODULE_PARM()(在头文件 linux/module.h)。运行时，insmod 将给变量赋予命令行的参数，如同 ./insmod mymodule.o myvariable=5。为使代码清晰，变量的声明和宏都应该放在 模块代码的开始部分。以下的代码范例也许将比我公认差劲的解说更好。

宏 MODULE_PARM()需要两个参数，变量的名字和其类型。支持的类型有" b": 比特型，"h": 短整型, "i": 整数型，" l: 长整型和 "s": 字符串型,其中正数型既可为 signed 也可为 unsigned。 字符串类型应该声明为"char *"这样 insmod 就可以为它们分配内存空间。你应该总是为你的变量赋初值。 这是内核编程，代码要编写的十分谨慎。举个例子：

```
int myint = 3;
char *mystr;

MODULE_PARM(myint, "i" );
MODULE_PARM(mystr, "s" );
```

数组同样被支持。在宏 MODULE_PARM 中在类型符号前面的整型值意味着一个指定了最大长度的数组。用'-'隔开的两个数字则分别意味着最小和最大长度。下面的例子中，就声明了一个最小长度为 2,最大长度为 4 的整形数组。

```
int myshortArray[4];
MODULE_PARM (myintArray, "3-9i" );
```

将初始值设为缺省使用的 IO 端口或 IO 内存是一个不错的作法。如果这些变量有缺省值，则可以进行自动设备检测， 否则保持当前设置的值。我们将在后续章节解释清楚相关内容。在这里我只是演示如何向一个模块传递参数。

最后，还有这样一个宏，MODULE_PARM_DESC()被用来注解该模块可以接收的参数。该宏 两个参数：变量名和一个格式自由的对该变量的描述。

Example 2-7. hello-5.c

```
/*
*  hello-5.c - Demonstrates command line argument passing to a module.
*/
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/stat.h>

MODULE_LICENSE("GPL" );
MODULE_AUTHOR(" Peter Jay Salzman" );

static short int myshort = 1;
static int myint = 420;
static long int mylong = 9999;
static char *mystring = "blah";

/*
* module_param(foo, int, 0000)
* The first param is the parameters name
* The second param is it's data type
* The final argument is the permissions bits,
* for exposing parameters in sysfs (if non-zero) at a later stage.
*/

module_param(myshort, short, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(myshort, "A short integer" );
module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(myint, "An integer" );
module_param(mylong, long, S_IRUSR);
MODULE_PARM_DESC(mylong, "A long integer" );
module_param(mystring, charp, 0000);
MODULE_PARM_DESC(mystring, "A character string" );

static int __init hello_5_init(void)
{
        printk(KERN_ALERT "Hello, world 5\n=============\n" );
        printk(KERN_ALERT "myshort is a short integer: %hd\n", myshort);
        printk(KERN_ALERT "myint is an integer: %d\n", myint);
        printk(KERN_ALERT "mylong is a long integer: %ld\n", mylong);
        printk(KERN_ALERT "mystring is a string: %s\n", mystring);
        return 0;
}

static void __exit hello_5_exit(void)
{
        printk(KERN_ALERT "Goodbye, world 5\n" );
}

module_init(hello_5_init);
module_exit(hello_5_exit);
```

```
satan# insmod hello-5.o mystring="bebop" mybyte=255 myintArray=-1
mybyte is an 8 bit integer: 255
myshort is a short integer: 1
myint is an integer: 20
mylong is a long integer: 9999
mystring is a string: bebop
myintArray is -1 and 420

satan# rmmod hello-5
Goodbye, world 5

satan# insmod hello-5.o mystring="supercalifragilisticexpialidocious" \
> mybyte=256 myintArray=-1,-1
mybyte is an 8 bit integer: 0
myshort is a short integer: 1
myint is an integer: 20
mylong is a long integer: 9999
mystring is a string: supercalifragilisticexpialidocious
myintArray is -1 and -1

satan# rmmod hello-5
Goodbye, world 5

satan# insmod hello-5.o mylong=hello
hello-5.o: invalid argument syntax for mylong: 'h'
```

由多个文件构成的内核模块

有时将模块的源代码分为几个文件是一个明智的选择。在这种情况下，你需要：

只要在一个源文件中添加#define __NO_VERSION__预处理命令。 这很重要因为 module.h 通常包含 kernel_version 的定义，此时一个存储着内核版本的全局变量将会被编译。但如果此时你又要包含头文件 version.h，你必须手动包含它，因为 module.h 不会再包含它如果打开预处理选项__NO_VERSION__。

像通常一样编译。

将所有的目标文件连接为一个文件。在 x86 平台下，使用命令 ld -m elf_i386 -r -o <module name.o> <1st src file.o> <2nd src file.o>。

此时 Makefile 一如既往会帮我们完成编译和连接的脏活。

这里是这样的一个模块范例。

Example 2-8. start.c

```
/*
*  start.c - Illustration of multi filed modules
*/

#include <linux/kernel.h>        /* We're doing kernel work */
#include <linux/module.h>         /* Specifically, a module */

int init_module(void)
{
        printk("Hello, world - this is the kernel speaking\n" );
        return 0;
}
```

另一个文件：

Example 2-9. stop.c

```
/*
*  stop.c - Illustration of multi filed modules
*/

#include <linux/kernel.h>        /* We're doing kernel work */
#include <linux/module.h>         /* Specifically, a module  */

void cleanup_module()
{
        printk("<1>Short is the life of a kernel module\n" );
}
```

最后是该模块的 Makefile：

Example 2-10. Makefile

```
obj-m += hello-1.o
obj-m += hello-2.o
obj-m += hello-3.o
obj-m += hello-4.o
obj-m += hello-5.o
obj-m += startstop.o
startstop-objs := start.o stop.o
```

为已编译的内核编译模块

很显然，我们强烈推荐你编译一个新的内核，这样你就可以打开内核中一些有用的排错功能，像强制卸载模块(MODULE_FORCE_UNLOAD)： 当该选项被打开时，你可以 rmmod -f module 强制内核卸载一个模块，即使内核认为这是不安全的。该选项可以为你节省不少开发时间。

但是，你仍然有许多使用一个正在运行中的已编译的内核的理由。例如，你没有编译和安装新内核的权限，或者你不希望重启你的机器来运行新内核。 如果你可以毫无阻碍的编译和使用一个新的内核，你可以跳过剩下的内容，权当是一个脚注。

如果你仅仅是安装了一个新的内核代码树并用它来编译你的模块，当你加载你的模块时，你很可能会得到下面的错误提示：

```
insmod: error inserting 'poet_atkm.ko': -1 Invalid module format
```

一些不那么神秘的信息被纪录在文件/var/log/messages 中；

```
Jun  4 22:07:54 localhost kernel: poet_atkm: version magic '2.6.5-1.358custom 686
REGPARM 4KSTACKS gcc-3.3' should be '2.6.5-1.358 686 REGPARM 4KSTACKS gcc-3.3'
```

换句话说，内核拒绝加载你的模块因为记载版本号的字符串不符(更确切的说是版本印戳)。版本印戳作为一个静态的字符串存在于内核模块中，以 vermagic: 。 版本信息是在连接阶段从文件 init/vermagic.o 中获得的。 查看版本印戳和其它在模块中的一些字符信息，可以使用下面的命令 modinfo module.ko：

```
[root@pcsenonsrv 02-HelloWorld]# modinfo hello-4.ko
license:        GPL
author:         Peter Jay Salzman <p@dirac.org>
description:    A sample driver
vermagic:       2.6.5-1.358 686 REGPARM 4KSTACKS gcc-3.3
depends:
```

我们可以借助选项--force-vermagic 解决该问题，但这种方法有潜在的危险，所以在成熟的模块中也是不可接受的。 解决方法是我们构建一个同我们预先编译好的内核完全相同的编译环境。如何具体实现将是该章后面的内容。

首先，准备同你目前的内核版本完全一致的内核代码树。然后，找到你的当前内核的编译配置文件。通常它可以在路径 /boot 下找到，使用像 config-2.6.x 的文件名。你可以直接将它拷贝到内核代码树的路径下： cp /boot/config-`uname -r` /usr/src/linux-`uname -r`/.config。

让我们再次注意一下先前的错误信息：仔细看的话你会发现，即使使用完全相同的配置文件，版本印戳还是有细小的差异的，但这足以导致 模块加载的失败。这其中的差异就是在模块中出现却不在内核中出现的 custom 字符串，是由某些发行版提供的修改过的 makefile 导致的。检查 /usr/src/linux/Makefile，确保下面这些特定的版本信息同你使用的内核完全一致：

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 5
EXTRAVERSION = -1.358custom
...
```

像上面的情况你就需要将 EXTRAVERSION 一项改为-1.358。我们的建议是将原始的 makefile 备份在 /lib/modules/2.6.5-1.358/build 下。一个简单的命令 cp /lib/modules/`uname -r`/build/Makefile /usr/src/linux-`uname -r`即可。另外，如果你已经在运行一个由上面的错误的 Makefile 编译的内核，你应该重新执行 make，或直接对应 /lib/modules/2.6.x/build/include/linux/version.h 从文件 /usr/src/linux-2.6.x/include/linux/version.h 修改 UTS_RELEASE，或用前者覆盖后者的。

现在，请执行 make 来更新设置和版本相关的头文件，目标文件：

```
[root@pcsenonsrv linux-2.6.x]# make
CHK     include/linux/version.h
UPD     include/linux/version.h
SYMLINK include/asm -> include/asm-i386
SPLIT   include/linux/autoconf.h -> include/config/*
HOSTCC  scripts/basic/fixdep
HOSTCC  scripts/basic/split-include
HOSTCC  scripts/basic/docproc
HOSTCC  scripts/conmakehash
HOSTCC  scripts/kallsyms
CC      scripts/empty.o
...
```

如果你不是确实想编译一个内核，你可以在 SPLIT 后通过按下 CTRL-C 中止编译过程。因为此时你需要的文件 已经就绪了。现在你可以返回你的模块目录然后编译加载它：此时模块将完全针对你的当前内核编译，加载时也不会由任何错误提示。

Chapter 3. Preliminaries

内核模块和用户程序的比较

内核模块是如何开始和结束的

用户程序通常从函数 main()开始，执行一系列的指令并且 当指令执行完成后结束程序。内核模块有一点不同。内核模块要么从函数 init_module 或是你用宏 module_init 指定的函数调用开始。这就是内核模块 的入口函数。它告诉内核模块提供那些功能扩展并且让内核准备好在需要时调用它。 当它完成这些后，该函数就执行结束了。模块在被内核调用前也什么都不做。

所有的模块或是调用 cleanup_module 或是你用宏 module_exit 指定的函数。这是模块的退出函数。它撤消入口函数所做的一切。 例如注销入口函数所注册的功能。

所有的模块都必须有入口函数和退出函数。既然我们有不只一种方法去定义这两个 函数，我将努力使用"入口函数"和"退出函数"来描述 它们。但是当我只用 init_module 和 cleanup_module 时，我希望你明白我指的是什么。

模块可调用的函数

程序员并不总是自己写所有用到的函数。一个常见的基本的例子就是 printf()你使用这些 C 标准库，libc 提供的库函数。 这些函数(像 printf()) 实际上在连接之前并不进入你的程序。 在连接时这些函数调用才会指向 你调用的库，从而使你的代码最终可以执行。

内核模块有所不同。在 hello world 模块中你也许已经注意到了我们使用的函数 printk() 却没有包含标准 I/O 库。这是因为模块是在 insmod 加 载时才连接的目标文件。那些要用到的函数的符号链接是内核自己提供的。 也就是说， 你可以在内核模块中使用的函数只能来自内核本身。如果你对内核提供了哪些函数符号 链接感兴趣，看一看文件/proc/kallsyms。

需要注意的一点是库函数和系统调用的区别。库函数是高层的，完全运行在用户空间， 为程序员提供调用真正的在幕后 完成实际事务的系统调用的更方便的接口。系统调用在内核 态运行并且由内核自己提供。标准 C 库函数 printf()可以被看做是一 个通用的输出语句，但它实际做的是将数据转化为符合格式的字符串并且调用系统调用 write()输出这些字符串。

是否想看一看 printf()究竟使用了哪些系统调用? 这很容易，编译下面的代码。

```
#include <stdio.h>
int main(void)
{ printf("hello" ); return 0; }
```

使用命令 gcc -Wall -o hello hello.c 编译。用命令 strace hello 行该可执行文件。是否很惊讶？ 每一行都和一个系统调用相对应。 strace[1] 是一个非常有用的程序，它可以告诉你程序使用了哪些系统调用和这些系统调用的参数，返回值。 这是一个极有价值的查看程序在干什么的工具。在输出的末尾，你应该看到这样类似的一行 write(1, "hello", 5hello)。这就是我们要找的。藏在面具 printf() 的真实面目。既然绝大多数人使用库函数来对文件 I/O 进行操作(像 fopen, fputs, fclose)。 你可以查看 man 说明的第二部分使用命令 man 2 write. 。man 说明的第二部分 专门介绍系统调用(像 kill()和 read())。man 说明的第三部分则专门介绍你可能更熟悉的库函数， (像 cosh()和 random())。

你甚至可以编写代码去覆盖系统调用，正如我们不久要做的。骇客常这样做来为系统安装后门或木马。但你可以用它来完成一些更有益的事，像让内核在每次某人删除文件时输出“Tee hee, that tickles!”的信息。

用户空间和内核空间

内核全权负责对硬件资源的访问，不管被访问的是显示卡，硬盘，还是内存。 用户程序常为这些资源竞争。就如同我在保存这 份文档同时本地数据库正在更新。 我的编辑器 vim 进程和数据库更新进程同时要求访问硬盘。内核必须使这些请求有条不紊的进行， 而不是随用户的意愿提供计算机资源。 为方便实现这种机制， CPU 可以在不同的状态运行。不同的状态赋予不同的你对系统操作的自由。Intel 80836 架构有四种状态。 Unix 只使用了其中 的两种，最高级的状态(操作状态 0,即“超级状态”，可以执行任何操作)和最低级的状态 (即“用户状态”)。

回忆以下我们对库函数和系统调用的讨论，一般库函数在用户态执行。 库函数调用一个或几个系统调用，而这些系统调用为库函数完成工作，但是在超级状态。 一旦系统调用完成工作后系统调用就返回同时程序也返回用户态。

命名空间

如果你只是写一些短小的 C 程序，你可为你的变量起一个方便的和易于理解的变量名。 但是，如果你写的代码只是 许多其它人写的代码的一部分，你的全局一些就会与其中的全局变量发生冲突。 另一个情况是一个程序中有太多的 难以理解的变量名，这又会导致变量命名空间污染 在大型项目中，必须努力记住保留的变量名，或为独一无二的命名使用一种统一的方法。

当编写内核代码时，即使是最小的模块也会同整个内核连接，所以这的确是个令人头痛的问题。 最好的解决方法是声明你的变量为 static 静态的并且为你的符号使用一个定义的很好的前缀。 传统中，使用小写字母的内核前缀。如果你不想将所有的东西都声明为 static 静态的， 另一个选择是声明一个 symbol table（符号表）并向内核注册。我们将在以后讨论。

文件/proc/kallsyms 保存着内核知道的所有的符号，你可以访问它们， 因为它们是内核代码空间的一部分。

代码空间

内存管理是一个非常复杂的课题。O'Reilly 的《Understanding The Linux Kernel》绝大部分都在 讨论内存管理！我们 并不准备专注于内存管理，但有一些东西还是得知道的。

如果你没有认真考虑过内存设计缺陷意味着什么，你也许会惊讶的获知一个指针并不指向一个确切 的内存区域。当一个进程建立时，内核为它分配一部分确切的实际内存空间并把它交给进程，被进程的 代码，变量，堆栈和其它一些计算机学的专家才明白的东西使用[2]。这些内存从0 开始并可以扩展到需要的地方。这些 内存空间并不重叠，所以即使进程访问同一个内存地址，例如 0xbffff978， 真实的物理内存地址其实是不同的。进程实际指向的是一块被分配的内存中以 0xbffff978 为偏移量的一块内存区域。绝大多数情况下，一个进程像普通的"Hello, World"不可以访问别的进程的 内存空间，尽管有实现这种机制的方法。 我们将在以后讨论。

内核自己也有内存空间。既然一个内核模块可以动态的从内核中加载和卸载，它其实是共享内核的 内存空间而不是自己拥有 独立的内存空间。因此，一旦你的模块具有内存设计缺陷，内核就是内存设计缺陷了。 如果你在错误的覆盖数据，那么你就在 破坏内核的代码。这比现在听起来的还糟。所以尽量小心谨慎。

顺便提一下，以上我所指出的对于任何单整体内核的操作系统都是真实的[3]。 也存在模块化微内核的操作系统，如 GNU Hurd 和 QNX Neutrino。

Device Drivers

一种内核模块是设备驱动程序，为使用硬件设备像电视卡和串口而编写。 在 Unix 中，任何设备都被当作路径/dev 的设备文件处理，并通过这些设备文件提供访问硬件的方法。 设备驱动为用户程序 访问硬件设备。举例来说，声卡设备驱动程序 es1370.o 将会把设备文件 /dev/sound 同声卡硬件 Ensoniq IS1370 联系起来。 这样用户程序像 mp3blaster 就可以通过访问设备文件/dev/sound 运行而不必知道那种声卡硬件安装在系统上。

Major and Minor Numbers
让我们来研究几个设备文件。这里的几个设备文件代表着一块主 IDE 硬盘上的头三个分区：

```
# ls -l /dev/hda[1-3]
brw-rw----  1 root  disk  3, 1 Jul  5  2000 /dev/hda1
brw-rw----  1 root  disk  3, 2 Jul  5  2000 /dev/hda2
brw-rw----  1 root  disk  3, 3 Jul  5  2000 /dev/hda3
```

注意一下被逗号隔开的两列。第一个数字被叫做主设备号，第二个被叫做从设备号。 主设备号决定使用何种设备驱动程序。 每种不同的设备都被分配了不同的主设备号； 所有具有相同主设备号的设备文件都是被同一个驱动程序控制。上面例子中的 主设备号都为 3， 表示它们都被同一个驱动程序控制。

从设备号用来区别驱动程序控制的多个设备。上面例子中的从设备号不相同是因为它们被识别为几个设备。

设备被大概的分为两类：字符设备和块设备。区别是块设备有缓冲区，所以它们可以对请求进行优化排序。 这对存储设备尤其 重要，因为读写相邻的文件总比读写相隔很远的文件要快。另一个区别是块设备输入和输出 都是以数据块为单位的，但是字符设备 就可以自由读写任意量的字节。大部分硬件设备为字符设备，因为它们 不需要缓冲区和数据不是按块来传输的。你可以通过命令 ls -l 输出的头一个字母识别一个 设备为何种设备。如果是'b' 就是块设备，如果是'c'就是字符设备。以上你看到的是块设备。这儿还有一些 字符设备文件（串口）：

```
crw-rw----  1 root  dial 4, 64 Feb 18 23:34 /dev/ttyS0
crw-r-----  1 root  dial 4, 65 Nov 17 10:26 /dev/ttyS1
crw-rw----  1 root  dial 4, 66 Jul  5  2000 /dev/ttyS2
crw-rw----  1 root  dial 4, 67 Jul  5  2000 /dev/ttyS3
```

如果你想看一下已分配的主设备号都是些什么设备可以看一下文件 /usr/src/linux/Documentation/devices.txt。

系统安装时，所有的这些设备文件都是由命令 mknod 建立的。去建立一个新的名叫 coffee'，主设备号为 12 和从设备号为 2 的设备文件，只要简单的 执行命令 mknod /dev/coffee c 12 2。你并不是必须将设备文件放在目录 /dev 中，这只是一个传统。Linus 本人是这样做的，所以你最好也不例外。 但是，当你测试一个模块时，在工作目录建立一个设备文件也不错。 只要保证完成后将它放在驱动程序找得到的地方。

我还想声明在以上讨论中隐含的几点。当系统访问一个系统文件时， 系统内核只使用主设备号来区别设备类型和决定使用何种内核模块。系统 内核并不需要知道从设备号。内核模块驱动本身才关注从设备号，并用之来 区别其操纵的不同设备。

另外，我这儿提到的硬件是比那种可以握在手里的 PCI 卡稍微抽象一点的东西。看一下下面的两个设备文件：

```
% ls -l /dev/fd0 /dev/fd0u1680
brwxrwxrwx   1 root  floppy    2,  0 Jul  5  2000 /dev/fd0
brw-rw----   1 root  floppy    2, 44 Jul  5  2000 /dev/fd0u1680
```

你现在立即明白这是快设备的设备文件并且它们是有相同的驱动内核模块来操纵 （主设备号都为 2)）。你也许也意识到它们都是你的软盘驱动器， 即使你实际上只有一个软盘驱动器。为什么是两个设备文件？因为它们其中的一个代表着 你的 1.44 MB 容量的软驱，另一个代表着你的 1.68 MB 容量的，被某些人称为"超级格式化"的软驱。 这就是一个不同的从设备号代表着相同硬件设备的例子。请清楚的意识到我们提到的硬件有时 可能是非常抽象的。

Notes

[1] 这是一个去跟踪程序究竟在做什么的非常有价值的工具。

[2] 我是物理专业的， 而不是主修计算机。

[3] 这不同于将所有的内核模块编译进内核，但意思确实是一样的。

Chapter 4. Character Device Files

字符设备文件

关于 file_operations 结构体

结构体 file_operations 在头文件 linux/fs.h 中定义，用来存储驱动内核模块提供的对 设备进行各种操作的函数的指针。该结构体的每个域都对应着驱动内核模块用来处理某个被请求的 事务的函数的地址。

举个例子，每个字符设备需要定义一个用来读取设备数据的函数。结构体 file_operations 中存储着内

```
struct file_operations {
        struct module *owner;
        loff_t(*llseek) (struct file *, loff_t, int);
        ssize_t(*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t(*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
        ssize_t(*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t(*aio_write) (struct kiocb *, const char __user *, size_t,
                             loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int,
                      unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t(*readv) (struct file *, const struct iovec *, unsigned long,
                         loff_t *);
        ssize_t(*writev) (struct file *, const struct iovec *, unsigned long,
                          loff_t *);
        ssize_t(*sendfile) (struct file *, loff_t *, size_t, read_actor_t,
                            void __user *);
        ssize_t(*sendpage) (struct file *, struct page *, int, size_t,
                            loff_t *, int);
        unsigned long (*get_unmapped_area) (struct file *, unsigned long,
                                            unsigned long, unsigned long,
                                            unsigned long);
};
```

核模块中执行这项操作的函数的地址。一下是该结构体 在内核 2.6.5 中看起来的样子：

驱动内核模块是不需要实现每个函数的。像视频卡的驱动就不需要从目录的结构 中读取数据。那么，相对应的 file_operations 重的项就为 NULL。

gcc 还有一个方便使用这种结构体的扩展。你会在较现代的驱动内核模块中见到。 新的使用这种结构体的方式如下：

```
struct file_operations fops = {
        read: device_read,
        write: device_write,
        open: device_open,
        release: device_release
};
```

同样也有 C99 语法的使用该结构体的方法，并且它比 GNU 扩展更受推荐。我使用的版本为 2.95 为了方便那些想移植你的代码的人，你最好使用这种语法。它将提高代码的兼容性：

```
struct file_operations fops = {
        .read = device_read,
        .write = device_write,
        .open = device_open,
        .release = device_release
};
```

这种语法很清晰，你也必须清楚的意识到没有显示声明的结构体成员都被 gcc 初始化为 NULL。

指向结构体 struct file_operations 的指针通常命名为 fops。

关于 file 结构体

每一个设备文件都代表着内核中的一个 file 结构体。该结构体在头文件 linux/fs.h 定义。注意，file 结构体是内核空间的结构体， 这意味着它不会在用户程序的代码中出现。它绝对不是在 glibc 中定义的 FILE。 FILE 自己也从不在内核空间的函数中出现。它的名字确实挺让人迷惑的。 它代表着一个抽象的打开的文件，但不是那种在磁盘上用结构体 inode 表示的文件。

指向结构体 struct file 的指针通常命名为 filp。 你同样可以看到 struct file file 的表达方式，但不要被它诱惑。

去看看结构体 file 的定义。大部分的函数入口，像结构体 struct dentry 没有被设备驱动模块使用，你大可忽略它们。这是因为设备驱动模块并不自己直接填充结构体 file：它们只是使用在别处建立的结构体 file 中的数据。

注册一个设备

如同先前讨论的，字符设备通常通过在路径/dev[1]设备文 件访问。主设备号告诉你哪些驱动模块是用来操纵哪些硬件设备的。从设备号是驱动模 块自己使用来区别它操纵的不同设备，当此驱动模块操纵不只一个设备时。

将内核驱动模块加载入内核意味着要向内核注册自己。这个工作是和驱动模块获 得主设备号时初始化一同进行的。你可以使用头文件 linux/fs.h 中的函数 register_chrdev 来实 现。

int register_chrdev(unsigned int major, const char *name, struct file_operations *fops);

其中 unsigned int major 是你申请的主设备号， const char *name 是将要在文件/proc/devices struct file_operations *fops 是指向你的驱动模块的 file_operations 表的指针。负的返回值意味着注册失败。 注意注册并不需要提供从设备号。内核本身并不在意从设备号。

现在的问题是你如何申请到一个没有被使用的主设备号？最简单的方法是查看文件 Documentation/devices.txt 从中挑选一个没有被使用的。这不是 一劳永逸的方法因为你无法得知该主设备号在将来会被占用。最终的方法是让内核为你动 态分配一个。

如果你向函数 register_chrdev 传递为 0 的主设备号，那么 返回的就是动态分配的主设备号。副作用就是既然你无法得知主设备号，你就无法预先建 立一个设备文件。 有多种解决方法。第一种方法是新注册的驱动模块会输出自己新分配 到的主设备号，所以我们可以手工建立需要的设备文件。第二种是利用文件 /proc/devices 新注册的驱动模块的入口，要么手工建立设备文件， 要么编一个脚本去自动读取该文件并且生成设备文件。第三种是在我们的模块中，当注册 成功时，使用 mknod 统调用建立设备文件并且调用 rm 删除该设备 文件在驱动模块调用函数 cleanup_module 前。

注销一个设备

即使时 root 也不能允许随意卸载内核模块。当一个进程已经打开一个设备文件时我 们卸载了该设备文件使用的内核模块，我们此时再对该文件的访问将会导致对已卸载的内 核模块代码内存区的访问。幸运的话我们最多获得一个讨厌的错误警告。如果此时已经在 该内存区加载了另一个模块，倒霉的你将会在内核中跳转执行意料外的代码。结果是无法 预料的，而且多半是不那么令人愉快的。

平常，当你不允许某项操作时，你会得到该操作返回的错误值（一般为一负的值）。 但对于无返回值的函数 cleanup_module 这是不可能的。然而，却有 一个计数器跟踪着有多少进程正在使用该模块。你可以通过查看文件 /proc/modules 的第三列来获取这些信息。如果该值非零，则卸载 就会失败。你不需要在你模块中的函数 cleanup_module 中检查该 计数器，因为该项检查由头文件 linux/module.c 中定义的系统调用 sys_delete_module 完成。你也不应该直接对该计数器进行操作。 你应该使用在文件 linux/modules.h 定义的宏 来增加，减小和读取该计数器：

try_module_get(THIS_MODULE): Increment the use count.

try_module_put(THIS_MODULE): Decrement the use count.

保持该计数器时刻精确是非常重要的；如果你丢失了正确的计数，你将无法卸载模块， 那就只有重启了。不过这种情况在今后编写内核模块时也是无法避免的。

chardev.c

下面的代码示范了一个叫做 chardev 的字符设备。你可以用 cat 输出该设备文件的内容（或用别的程序打开它）时，驱动模块 会将该设备文件被读取的次数显示。目前对设备文件的写操作还不被支持（像 echo "hi" > /dev/hello），但会捕捉这些操作并且告诉用户该操作不被支持。不要担心我 们对读入缓冲区的数据做了什么；我们什么都没做。我们只是读入数据并输出我们已经接 收到的数据的信息。

Example 4-1. chardev.c

```
/*
*  chardev.c: Creates a read-only char device that says how many times
*  you've read from the dev file
*/

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>        /* for put_user */

/*
*  Prototypes - this would normally go in a .h file
*/
int init_module(void);
void cleanup_module(void);
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);

#define SUCCESS 0
#define DEVICE_NAME "chardev"         /* Dev name as it appears in /proc/devices   */
#define BUF_LEN 80                    /* Max length of the message from the device */

/*
* Global variables are declared as static, so are global within the file.
*/

static int Major;                 /* Major number assigned to our device driver */
static int Device_Open = 0;         /* Is device open?
                                 * Used to prevent multiple access to device */
static char msg[BUF_LEN];         /* The msg the device will give when asked */
static char *msg_Ptr;
```

```
static struct file_operations fops = {
        .read = device_read,
        .write = device_write,
        .open = device_open,
        .release = device_release
};

/*
* Functions
*/

int init_module(void)
{
        Major = register_chrdev(0, DEVICE_NAME, &fops);

        if (Major < 0) {
                printk("Registering the character device failed with %d\n",
                      Major);
                return Major;
        }

        printk("<1>I was assigned major number %d.  To talk to\n", Major);
        printk("<1>the driver, create a dev file with\n");
        printk("'mknod /dev/hello c %d 0'.\n", Major);
        printk("<1>Try various minor numbers.  Try to cat and echo to\n");
        printk("the device file.\n");
        printk("<1>Remove the device file and module when done.\n");

        return 0;
}

void cleanup_module(void)
{
        /*
         * Unregister the device
         */
        int ret = unregister_chrdev(Major, DEVICE_NAME);
        if (ret < 0)
                printk("Error in unregister_chrdev: %d\n", ret);
}
```

```
/*
* Methods
*/

/*
* Called when a process tries to open the device file, like
* "cat /dev/mycharfile"
*/
static int device_open(struct inode *inode, struct file *file)
{
        static int counter = 0;
        if (Device_Open)
                return -EBUSY;
        Device_Open++;
        sprintf(msg, "I already told you %d times Hello world!\n", counter++);
        msg_Ptr = msg;
        try_module_get(THIS_MODULE);

        return SUCCESS;
}

/*
* Called when a process closes the device file.
*/
static int device_release(struct inode *inode, struct file *file)
{
        Device_Open--;                /* We're now ready for our next caller */

        /*
         * Decrement the usage count, or else once you opened the file, you'll
         * never get get rid of the module.
         */
        module_put(THIS_MODULE);

        return 0;
}
```

```
/*
* Called when a process, which already opened the dev file, attempts to
* read from it.
*/
static ssize_t device_read(struct file *filp,   /* see include/linux/fs.h   */
                           char *buffer,    /* buffer to fill with data */
                           size_t length,   /* length of the buffer     */
                           loff_t * offset)
{
        /*
         * Number of bytes actually written to the buffer
         */
        int bytes_read = 0;

        /*
         * If we're at the end of the message,
         * return 0 signifying end of file
         */
        if (*msg_Ptr == 0)
                return 0;

        /*
         * Actually put the data into the buffer
         */
        while (length && *msg_Ptr) {

                /*
                 * The buffer is in the user data segment, not the kernel
                 * segment so "*" assignment won't work.  We have to use
                 * put_user which copies data from the kernel data segment to
                 * the user data segment.
                 */
                put_user(*(msg_Ptr++), buffer++);

                length--;
                bytes_read++;
        }

        /*
         * Most read functions return the number of bytes put into the buffer
         */
        return bytes_read;
}
```

```
/*
* Called when a process writes to dev file: echo "hi" > /dev/hello
*/
static ssize_t
device_write(struct file *filp, const char *buff, size_t len, loff_t * off)
{
        printk("<1>Sorry, this operation isn't supported.\n");
        return -EINVAL;
}
```

为多个版本的内核编写内核模块

系统调用，也就是内核提供给进程的接口，基本上是保持不变的。也许会添入新的 系统调用，但那些已有的不会被改动。这对于向下兼容是非常重要的。在多数情况下，设 备文件是保持不变的。但内核的内部在不同版本之间还是会有区别的。

Linux 内核分为稳定版本（版本号中间为偶数）和试验版本（版本号中间为奇数）。 试验版本中可以试验各种各样的新而酷的主意，有些会被证实是一个错误，有些在下一版 中会被完善。总之，你不能依赖这些版本中的接口（这也是我不在本文档中支持它们的原因， 它们更新的太快了)。在稳定版本中，我们可以期望接口保持一致，除了那些修改代码中错误的版本。

如果你要支持多版本的内核，你需要编写为不同内核编译的代码树。可以通过比较宏 LINUX_VERSION_CODE 和宏 KERNEL_VERSION 在版本号为 a.b.c 的内核中，该宏的值应该为 2^16×a+2^8×b+c

在上一个版本中该文档还保留了详细的如何向后兼容老内核的介绍，现在我们决定打破这个传统。 对为老内核编写驱动感兴趣的读者应该参考对应版本的 LKMPG，也就是说，2.4.x 版本的 LKMPG 对应 2.4.x 的内核，2.6.x 版本的 LKMPG 对应 2.6.x 的内核。

Notes

[1] 这只是习惯上的。将设备文件放 在你的用户目录下是没有问题的。但是当真正提供成熟的驱动模块时，请保证将设备文 件放在/dev 下。

Chapter 5. The /proc File System

关于 /proc 文件系统

在 Linux 中有另一种内核和内核模块向进程传递信息的方法，那就是通过 /proc 文件系统。它原先设计的目的是为查看进程信息 提供一个方便的途径，现在它被用来向用户提供各种内核中被感兴趣的内容。像文件 /proc/modules 里是已加载模块的列表，文件/proc/meminfo 里是关于内存使用的信息。

使用 proc 文件系统的方法同使用设备文件很相似。你建立一个包含 /proc 文件需要的所有信息的结构体， 这其中包括处理各种事务的函数的指针（在我们的例子中，只用到从/proc 文件读取信息的函数）。然后在 init_module 时向内核注册这个结构体，在 cleanup_module 时注销这个结构体。

我们使用 proc_register_dynamic[1]的原因是我们不用去设置 inode，而留给 内核去自动分配从而避免系统冲突错误。 普通的文件系统是建立在磁盘上的，而 /proc 的文件仅仅是建立在内存中的。 在前种情况中，inode 的数值是一个指向存储在磁盘某个位置的文件的索引节点（inode 就是 index-node 的缩写）。 该索引节点储存着文件的信息，像文件的权限；同时还有在哪儿能找到文件中的数据。

因为我们无法得知该文件是被打开的或关闭的，我们也无法去使用宏 try_module_get 和 try_module_put 在下面的模块中， 我们无法避免该文件被打开而同时模块又被卸载。在下章中我将介绍一个较难实现，却更灵活，更安全的处理 /proc 文件的方法。

Example 5-1. procfs.c

```
/*
*  procfs.c -  create a "file" in /proc
*/

#include <linux/module.h>         /* Specifically, a module */
#include <linux/kernel.h>         /* We're doing kernel work */
#include <linux/proc_fs.h>         /* Necessary because we use the proc fs */

struct proc_dir_entry *Our_Proc_File;

/* Put data into the proc fs file.
*
* Arguments
* =========
* 1. The buffer where the data is to be inserted, if
*    you decide to use it.
* 2. A pointer to a pointer to characters. This is
*    useful if you don't want to use the buffer
*    allocated by the kernel.
* 3. The current position in the file
```

```
* 4. The size of the buffer in the first argument.
* 5. Write a "1" here to indicate EOF.
* 6. A pointer to data (useful in case one common
*     read for multiple /proc/... entries)
*
* Usage and Return Value
* ======================
* A return value of zero means you have no further
* information at this time (end of file). A negative
* return value is an error condition.
*
* For More Information
* ====================
* The way I discovered what to do with this function
* wasn't by reading documentation, but by reading the
* code which used it. I just looked to see what uses
* the get_info field of proc_dir_entry struct (I used a
* combination of find and grep, if you're interested),
* and I saw that  it is used in <kernel source
* directory>/fs/proc/array.c.
*
* If something is unknown about the kernel, this is
* usually the way to go. In Linux we have the great
* advantage of having the kernel source code for
* free - use it.
*/
```

```
ssize_t
procfile_read(char *buffer,
              char **buffer_location,
              off_t offset, int buffer_length, int *eof, void *data)
{
        printk(KERN_INFO "inside /proc/test : procfile_read\n" );

        int len = 0;                /* The number of bytes actually used */
        static int count = 1;


        /*
         * We give all of our information in one go, so if the user asks us if we have more information the answer should always be no.
         *
         * This is important because the standard read function from the library would continue to issue the read system call until the kernel replies that it has no more information, or until its
         * buffer is filled.
         */
        if (offset > 0) {
                printk(KERN_INFO "offset %d : /proc/test : procfile_read, \
                      wrote %d Bytes\n", (int)(offset), len);
                *eof = 1;
                return len;
        }

        /*
         * Fill the buffer and get its length
         */
        len = sprintf(buffer,
                      "For the %d%s time, go away!\n", count,
                      (count % 100 > 10 && count % 100 < 14) ? "th" :
                      (count % 10 == 1) ? "st" :
                      (count % 10 == 2) ? "nd" :
                      (count % 10 == 3) ? "rd" : "th" );
        count++;

        /*
         * Return the length
         */
        printk(KERN_INFO
               "leaving /proc/test : procfile_read, wrote %d Bytes\n", len);
        return len;
}
```

```
int init_module()
{
        int rv = 0;
        Our_Proc_File = create_proc_entry("test", 0644, NULL);
        Our_Proc_File->read_proc = procfile_read;
        Our_Proc_File->owner = THIS_MODULE;
        Our_Proc_File->mode = S_IFREG | S_IRUGO;
        Our_Proc_File->uid = 0;
        Our_Proc_File->gid = 0;
        Our_Proc_File->size = 37;

        printk(KERN_INFO "Trying to create /proc/test:\n" );

        if (Our_Proc_File == NULL) {
                rv = -ENOMEM;
                remove_proc_entry("test", &proc_root);
                printk(KERN_INFO "Error: Could not initialize /proc/test\n" );
        } else {
                printk(KERN_INFO "Success!\n" );
        }

        return rv;
}

void cleanup_module()
{
        remove_proc_entry("test", &proc_root);
        printk(KERN_INFO "/proc/test removed\n" );
}
```

Notes

[1] 这是在 2.0 版本中的做法， 在版本 2.2 中，当我们把 inode 设为 0 时，就已经这样自动处理了。

Chapter 6. Using /proc For Input

使用 /proc 作为输入

现在我们有两种从内核模块获得输出的方法：我们可以注册一个设备驱动并用 mknod 生成一个设备文件，或者我们可以建立一个 /proc 文件。这样内核就可以告诉我们重要的信息。 剩下的唯一问题是我们没法反馈信息。第一种方法是向/proc 文件系统写入信息。

由于 /proc 文件系统是为内核输出其运行信息而设计的，它并未向内核输入信息提供了任何准备。 结构体 struct proc_dir_entry 并没有指向输入函数的指针，而是指向了一个输出函数。 作为替代办法，向/proc 写入信息，我们可以使用标准的文件系统提供的机制。

在 Linux 中有一种标准的注册文件系统的方法。既然每种文件系统都必须有处理文件 索引节点 inode 和文件本身的函数[1], 那么就一定有种结构体去存放这些函数的指针。这就是结构体 struct inode_operations， 它其中又包含一个指向结构体 struct file_operations 的指针。在 /proc 文件系统中， 当我们需要注册一个新文件时，我们被允许选择哪一个 struct inode_operations 结构体。这就是我们将使用的机制，用包含结构体 struct inode_operations 指针的结构体 struct file_operations 来指向我们的 module_input 和 module_output 函数。

需要注意的是"读"和"写"的含义在内核中是反过来的。"读"意味着输出，而"写"意味着输入。 这是从用户的角度来看待问题的。如果一个进程只能从内核的"输出"获得输入， 而内核也是从进程的输出中得到"输入"的。

在这儿另一件有趣的事就是 module_permission 函数了。该函数在每个进程想要对 /proc 文件系统内的文件操作时被调用，它来决定是否操作被允许。 目前它只是对操作和操作所属用户的 UID 进行判断，但它可以也把其它的东西包括进来， 像还有哪些别的进程在对该文件进行操作，当前的时间，或是我们最后接收到的输入。

加入宏 put_user 和 get_user 的原因是 Linux 的内存是使用分页机制的（在 Intel 架构下是如此，但其它架构下有可能不同）。 这就意味着指针自身并不是指向一个确实的物理内存地址，而知是分页中的一个地址， 而且你必须知道哪些分页将来是可用的。其中内核本身占用一个分页，其它的每个进程都有自己的分页。

进程能看得到的分页只有属于它自己的，所以当编写用户程序时，不用考虑分页的存在。 但是当你编写内核模块时，你就会访问由系统自动管理的内核所在的分页。 当一块内存缓冲区中的内容要在当前运行中的进程和内核之间传递时，内核的函数就接收指向在进程分页中的该内存缓冲区的指针。宏 put_user 和 get_user 允许你进行这样的访问内存的操作。

Example 6-1. procfs.c

```
/*
*  procfs.c -  create a "file" in /proc, which allows both input and output.
*/
#include <linux/kernel.h>         /* We're doing kernel work */
#include <linux/module.h>          /* Specifically, a module */
#include <linux/proc_fs.h>         /* Necessary because we use proc fs */
#include <asm/uaccess.h>           /* for get_user and put_user */

/*
* Here we keep the last message received, to prove
* that we can process our input
*/
#define MESSAGE_LENGTH 80
static char Message[MESSAGE_LENGTH];
static struct proc_dir_entry *Our_Proc_File;

#define PROC_ENTRY_FILENAME "rw_test"

static ssize_t module_output(struct file *filp,         /* see include/linux/fs.h    */
                             char *buffer,          /* buffer to fill with data */
                             size_t length,          /* length of the buffer     */
                             loff_t * offset)
```

```
{
        static int finished = 0;
        int i;
        char message[MESSAGE_LENGTH + 30];

        /*
         * We return 0 to indicate end of file, that we have
         * no more information. Otherwise, processes will
         * continue to read from us in an endless loop.
         */
        if (finished) {
                finished = 0;
                return 0;
        }

        /*
         * We use put_user to copy the string from the kernel's
         * memory segment to the memory segment of the process
         * that called us. get_user, BTW, is
         * used for the reverse.
         */
        sprintf(message, "Last input:%s", Message);
        for (i = 0; i < length && message[i]; i++)
                put_user(message[i], buffer + i);

        /*
         * Notice, we assume here that the size of the message
         * is below len, or it will be received cut. In a real
         * life situation, if the size of the message is less
         * than len then we'd return len and on the second call
         * start filling the buffer with the len+1'th byte of
         * the message.
         */
        finished = 1;

        return i;               /* Return the number of bytes "read" */
}
```

```
static ssize_t
module_input(struct file *filp, const char *buff, size_t len, loff_t * off)
{
        int i;
        /*
         * Put the input into Message, where module_output
         * will later be able to use it
         */
        for (i = 0; i < MESSAGE_LENGTH - 1 && i < len; i++)
                get_user(Message[i], buff + i);

        Message[i] = '\0';          /* we want a standard, zero terminated string */
        return i;
}

/*
* This function decides whether to allow an operation
* (return zero) or not allow it (return a non-zero
* which indicates why it is not allowed).
*
* The operation can be one of the following values:
* 0 - Execute (run the "file" - meaningless in our case)
* 2 - Write (input to the kernel module)
* 4 - Read (output from the kernel module)
*
* This is the real function that checks file
* permissions. The permissions returned by ls -l are
* for referece only, and can be overridden here.
*/
```

```
static int module_permission(struct inode *inode, int op, struct nameidata *foo)
{
        /*
         * We allow everybody to read from our module, but
         * only root (uid 0) may write to it
         */
        if (op == 4 || (op == 2 && current->euid == 0))
                return 0;

        /*
         * If it's anything else, access is denied
         */
        return -EACCES;
}

/*
* The file is opened - we don't really care about
* that, but it does mean we need to increment the
* module's reference count.
*/
int module_open(struct inode *inode, struct file *file)
{
        try_module_get(THIS_MODULE);
        return 0;
}

/*
* The file is closed - again, interesting only because
* of the reference count.
*/
int module_close(struct inode *inode, struct file *file)
{
        module_put(THIS_MODULE);
        return 0;               /* success */
}

static struct file_operations File_Ops_4_Our_Proc_File = {
        .read = module_output,
        .write = module_input,
        .open = module_open,
        .release = module_close,
};
```

```
/*
* Inode operations for our proc file. We need it so
* we'll have some place to specify the file operations
* structure we want to use, and the function we use for
* permissions. It's also possible to specify functions
* to be called for anything else which could be done to
* an inode (although we don't bother, we just put
* NULL).
*/

static struct inode_operations Inode_Ops_4_Our_Proc_File = {
        .permission = module_permission,        /* check for permissions */
};

/*
* Module initialization and cleanup
*/
int init_module()
{
        int rv = 0;
        Our_Proc_File = create_proc_entry(PROC_ENTRY_FILENAME, 0644, NULL);
        Our_Proc_File->owner = THIS_MODULE;
        Our_Proc_File->proc_iops = &Inode_Ops_4_Our_Proc_File;
        Our_Proc_File->proc_fops = &File_Ops_4_Our_Proc_File;
        Our_Proc_File->mode = S_IFREG | S_IRUGO | S_IWUSR;
        Our_Proc_File->uid = 0;
        Our_Proc_File->gid = 0;
        Our_Proc_File->size = 80;

        if (Our_Proc_File == NULL) {
                rv = -ENOMEM;
                remove_proc_entry(PROC_ENTRY_FILENAME, &proc_root);
                printk(KERN_INFO "Error: Could not initialize /proc/test\n");
        }

        return rv;
}

void cleanup_module()
{
        remove_proc_entry(PROC_ENTRY_FILENAME, &proc_root);
}
```

还需要更多的关于 procfs 的例子？我要提醒你的是：第一，有消息说也许不久 procfs 将被 sysfs 取代；第二， 如果你真的很想多了解些 procfs，你可以参考路径 linux/Documentation/DocBook/ 下的 那些技术性的文档。在内核代码树根目录下使用 make help 来获得如何将这些文档转化为你偏好的格式，例如： make htmldocs 。如果你要为内核加入一些你的文档，你也应该考虑这样做。

Notes

[1] 两者的区别是文件的操作针对具体的，实在的文件， 而文件索引节点的操作是针对文件的引用，像建立文件的连接等。

Chapter 7. Talking To Device Files

与设备文件对话 (writes and IOCTLs)

设备文件是用来代表相对应的硬件设备。绝大多数的硬件设备是用来进行输出和输入操作的， 所以在内核中肯定有内核从进程中获得发送到设备的输出的机制。这是通过打开一个设备文件然后 向其中进行写操作来实现的，如同对普通文件的写操作。在下面的的例子中，这是通过 device_write 实现的。

但这并不总是够用。设想你有一个通过串口连接的调制解调器(即使你使用的是内置调制解调器， 对于 CPU 来说同样也是通过连接在串口上来实现工作的)。通常我们通过打开一个设备文件向调制解调器 发送信息（将要通过通信线路传输的指令或数据）或读取信息（从通信线路中返回的响应指令或数据）。 但是，我们如何设置同串口对话的速率，也就是向串口传输数据的速率这个问题仍然没有解决。

解决之道是在 Unix 系统中的函数 ioctl(Input Output ConTroL 的简写)。每个设备可以有自己的 ioctl 命令，通过读取 ioctl's 可以从进程中向内核发送信息，或写 ioctl's 向进程返回信息 [1]，或者两者都是，或都不是。函数 ioctl 调用时需要三个参数：合适的设备文件的文件描述符，ioctl 号，和一个可以被一个任务使用来 传递任何东西的 long 类型的参数[2]

ioctl 号是反映主设备号，ioctl 的种类，对应的命令和参数类型的数字。它通常是通过在头文件中宏调用 (_IO, _IOR, _IOW 或_IOWR，取决于其种类)来建立的。该头文件应该被使用 ioctl 的用户程序包含（这样它们就可以生成正确的 ioctl's） 和内核驱动模块包含（这样模块才能理解它)。在下面的例子中，头文件为 chardev.h，源程序为 ioctl.c。

即使你只想在自己的模块中使用 ioctls，你最好还是接收正式的 ioctl 标准，这样当你意外的使用别人的 ioctls， 或别人使用你的时，你会知道有错误发生。详情参见内核代码目录树下的文件 Documentation/ioctl-number.txt.

Example 7-1. chardev.c

```
/*
*  chardev.c - Create an input/output character device
*/

#include <linux/kernel.h>         /* We're doing kernel work */
#include <linux/module.h>          /* Specifically, a module */
#include <linux/fs.h>
#include <asm/uaccess.h>          /* for get_user and put_user */

#include "chardev.h"
#define SUCCESS 0
#define DEVICE_NAME "char_dev"
#define BUF_LEN 80
```

```
/*
* Is the device open right now? Used to prevent
* concurent access into the same device
*/
static int Device_Open = 0;

/*
* The message the device will give when asked
*/
static char Message[BUF_LEN];

/*
* How far did the process reading the message get?
* Useful if the message is larger than the size of the
* buffer we get to fill in device_read.
*/
static char *Message_Ptr;

/*
* This is called whenever a process attempts to open the device file
*/
static int device_open(struct inode *inode, struct file *file)
{
#ifdef DEBUG
        printk("device_open(%p)\n", file);
#endif

        /*
         * We don't want to talk to two processes at the same time
         */
        if (Device_Open)
                return -EBUSY;

        Device_Open++;
        /*
         * Initialize the message
         */
        Message_Ptr = Message;
        try_module_get(THIS_MODULE);
        return SUCCESS;
}
```

```
static int device_release(struct inode *inode, struct file *file)
{
#ifdef DEBUG
        printk("device_release(%p,%p)\n", inode, file);
#endif

        /*
         * We're now ready for our next caller
         */
        Device_Open--;

        module_put(THIS_MODULE);
        return SUCCESS;
}

/*
* This function is called whenever a process which has already opened the
* device file attempts to read from it.
*/
static ssize_t device_read(struct file *file,          /* see include/linux/fs.h    */
                           char __user * buffer,          /* buffer to be
                                                        * filled with data */
                           size_t length,          /* length of the buffer      */
                           loff_t * offset)
```

```
{
        /*
         * Number of bytes actually written to the buffer
         */
        int bytes_read = 0;

#ifdef DEBUG
        printk("device_read(%p,%p,%d)\n", file, buffer, length);
#endif

        /*
         * If we're at the end of the message, return 0
         * (which signifies end of file)
         */
        if (*Message_Ptr == 0)
                return 0;

        /*
         * Actually put the data into the buffer
         */
        while (length && *Message_Ptr) {

                /*
                 * Because the buffer is in the user data segment,
                 * not the kernel data segment, assignment wouldn't
                 * work. Instead, we have to use put_user which
                 * copies data from the kernel data segment to the
                 * user data segment.
                 */
                put_user(*(Message_Ptr++), buffer++);
                length--;
                bytes_read++;
        }

#ifdef DEBUG
        printk("Read %d bytes, %d left\n", bytes_read, length);
#endif
```

```
        /*
         * Read functions are supposed to return the number
         * of bytes actually inserted into the buffer
         */
        return bytes_read;
}

/*
* This function is called when somebody tries to
* write into our device file.
*/
static ssize_t
device_write(struct file *file,
             const char __user * buffer, size_t length, loff_t * offset)
{
        int i;

#ifdef DEBUG
        printk("device_write(%p,%s,%d)", file, buffer, length);
#endif

        for (i = 0; i < length && i < BUF_LEN; i++)
                get_user(Message[i], buffer + i);

        Message_Ptr = Message;

        /*
         * Again, return the number of input characters used
         */
        return i;
}

/*
* This function is called whenever a process tries to do an ioctl on our
* device file. We get two extra parameters (additional to the inode and file
* structures, which all device functions get): the number of the ioctl called
* and the parameter given to the ioctl function.
*
* If the ioctl is write or read/write (meaning output is returned to the
* calling process), the ioctl call returns the output of this function.
*
*/
```

```
int device_ioctl(struct inode *inode,          /* see include/linux/fs.h */
                 struct file *file,          /* ditto */
                 unsigned int ioctl_num,          /* number and param for ioctl */
                 unsigned long ioctl_param)
{
        int i;
        char *temp;
        char ch;

        /*
         * Switch according to the ioctl called
         */
        switch (ioctl_num) {
        case IOCTL_SET_MSG:
                /*
                 * Receive a pointer to a message (in user space) and set that
                 * to be the device's message.  Get the parameter given to
                 * ioctl by the process.
                 */
                temp = (char *)ioctl_param;

                /*
                 * Find the length of the message
                 */
                get_user(ch, temp);
                for (i = 0; ch && i < BUF_LEN; i++, temp++)
                        get_user(ch, temp);

                device_write(file, (char *)ioctl_param, i, 0);
                break;

        case IOCTL_GET_MSG:
                /*
                 * Give the current message to the calling process -
                 * the parameter we got is a pointer, fill it.
                 */
                i = device_read(file, (char *)ioctl_param, 99, 0);
```

```
                /*
                 * Put a zero at the end of the buffer, so it will be
                 * properly terminated
                 */
                put_user('\0', (char *)ioctl_param + i);
                break;

        case IOCTL_GET_NTH_BYTE:
                /*
                 * This ioctl is both input (ioctl_param) and
                 * output (the return value of this function)
                 */
                return Message[ioctl_param];
                break;
        }

        return SUCCESS;
}

/* Module Declarations */

/*
* This structure will hold the functions to be called
* when a process does something to the device we
* created. Since a pointer to this structure is kept in
* the devices table, it can't be local to
* init_module. NULL is for unimplemented functions.
*/
struct file_operations Fops = {
        .read = device_read,
        .write = device_write,
        .ioctl = device_ioctl,
        .open = device_open,
        .release = device_release,        /* a.k.a. close */
};

/*
* Initialize the module - Register the character device
*/
```

```
int init_module()
{
        int ret_val;
        /*
         * Register the character device (atleast try)
         */
        ret_val = register_chrdev(MAJOR_NUM, DEVICE_NAME, &Fops);

        /*
         * Negative values signify an error
         */
        if (ret_val < 0) {
                printk("%s failed with %d\n",
                      "Sorry, registering the character device ", ret_val);
                return ret_val;
        }

        printk("%s The major device number is %d.\n",
              "Registeration is a success", MAJOR_NUM);
        printk("If you want to talk to the device driver,\n" );
        printk("you'll have to create a device file. \n" );
        printk("We suggest you use:\n" );
        printk("mknod %s c %d 0\n", DEVICE_FILE_NAME, MAJOR_NUM);
        printk("The device file name is important, because\n" );
        printk("the ioctl program assumes that's the\n" );
        printk("file you'll use.\n" );

        return 0;
}
```

```
/*
* Cleanup - unregister the appropriate file from /proc
*/
void cleanup_module()
{
        int ret;

        /*
         * Unregister the device
         */
        ret = unregister_chrdev(MAJOR_NUM, DEVICE_NAME);

        /*
         * If there's an error, report it
         */
        if (ret < 0)
                printk("Error in module_unregister_chrdev: %d\n", ret);
}
```

Example 7-2. chardev.h

```
/*
*  chardev.h - the header file with the ioctl definitions.
*
*  The declarations here have to be in a header file, because
*  they need to be known both to the kernel module
*  (in chardev.c) and the process calling ioctl (ioctl.c)
*/

#ifndef CHARDEV_H
#define CHARDEV_H

#include <linux/ioctl.h>

/*
* The major device number. We can't rely on dynamic
* registration any more, because ioctls need to know
* it.
*/
#define MAJOR_NUM 100

/*
* Set the message of the device driver
*/
```

```
#define IOCTL_SET_MSG _IOR(MAJOR_NUM, 0, char *)
/*
* _IOR means that we're creating an ioctl command
* number for passing information from a user process
* to the kernel module.
*
* The first arguments, MAJOR_NUM, is the major device
* number we're using.
*
* The second argument is the number of the command
* (there could be several with different meanings).
*
* The third argument is the type we want to get from
* the process to the kernel.
*/

/*
* Get the message of the device driver
*/
#define IOCTL_GET_MSG _IOR(MAJOR_NUM, 1, char *)
/*
* This IOCTL is used for output, to get the message
* of the device driver. However, we still need the
* buffer to place the message in to be input,
* as it is allocated by the process.
*/

/*
* Get the n'th byte of the message
*/
#define IOCTL_GET_NTH_BYTE _IOWR(MAJOR_NUM, 2, int)
/*
* The IOCTL is used for both input and output. It
* receives from the user a number, n, and returns
* Message[n].
*/

/*
* The name of the device file
*/
#define DEVICE_FILE_NAME "char_dev"
```

```
#endif
Example 7-3. ioctl.c

/*
*  ioctl.c - the process to use ioctl's to control the kernel module
*
*  Until now we could have used cat for input and output.  But now
*  we need to do ioctl's, which require writing our own process.
*/

/*
* device specifics, such as ioctl numbers and the
* major device file.
*/
#include "chardev.h"

#include <fcntl.h>                 /* open */
#include <unistd.h>                  /* exit */
#include <sys/ioctl.h>                  /* ioctl */

/*
* Functions for the ioctl calls
*/

ioctl_set_msg(int file_desc, char *message)
{
        int ret_val;

        ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);

        if (ret_val < 0) {
                printf("ioctl_set_msg failed:%d\n", ret_val);
                exit(-1);
        }
}
```

```
ioctl_get_msg(int file_desc)
{
        int ret_val;
        char message[100];

        /*
         * Warning - this is dangerous because we don't tell
         * the kernel how far it's allowed to write, so it
         * might overflow the buffer. In a real production
         * program, we would have used two ioctls - one to tell
         * the kernel the buffer length and another to give
         * it the buffer to fill
         */
        ret_val = ioctl(file_desc, IOCTL_GET_MSG, message);

        if (ret_val < 0) {
                printf("ioctl_get_msg failed:%d\n", ret_val);
                exit(-1);
        }

        printf("get_msg message:%s\n", message);
}

ioctl_get_nth_byte(int file_desc)
{
        int i;
        char c;

        printf("get_nth_byte message:" );

        i = 0;
        while (c != 0) {
                c = ioctl(file_desc, IOCTL_GET_NTH_BYTE, i++);

                if (c < 0) {
                        printf
                           ("ioctl_get_nth_byte failed at the %d'th byte:\n",
                            i);
                        exit(-1);
                }

                putchar(c);
        }
        putchar('\n');
}
```

```
/*
* Main - Call the ioctl functions
*/
main()
{
        int file_desc, ret_val;
        char *msg = "Message passed by ioctl\n";

        file_desc = open(DEVICE_FILE_NAME, 0);
        if (file_desc < 0) {
                printf("Can't open device file: %s\n", DEVICE_FILE_NAME);
                exit(-1);
        }

        ioctl_get_nth_byte(file_desc);
        ioctl_get_msg(file_desc);
        ioctl_set_msg(file_desc, msg);

        close(file_desc);
}
```

Notes

[1] 注意这儿"读"与"写"的角色再次翻转过来，在 ioctl's 中读是向内核发送信息， 而写是从内核获取信息。

[2] 这样的表述并不准确。 例如你不能在 ioctl 中传递一个结构体，但你可以通过传递指向这个结构体的指针实现。

Chapter 8. System Calls

系统调用

到目前为止，我们所做的只是使用完善的内核机制注册/proc 文件和处理设备的对象。如果只是想写一个设备驱动， 这些内核程序员设定的方式已经足够了。但是，你不想做一些不寻常的事吗， 想使你的系统看起来不一样吗？当然，这取决你自己。

这里可是一个危险的地方。下面的这个例子中，我关闭了系统调用 open()。这意味着我无法打开任何文件，执行任何程序，连使用 shutdown 关机都不行，关机只能靠摁电源按钮了。幸运的话，不会有文件丢失。 要保证不丢失文件的话，在 insmod 和 rmmod 之前请执行 sync 命令。

别管什么/proc 文件和什么设备文件了， 它们只是小的细节问题。所有进程同内核打交道的根本方式是系统调用。 当一个进程需要内核提供某项服务时（像打开一个文件，生成一个新进程，或要求更多的内存）， 就会发生系统调用。如果你想你的系统运作方式看起来有意思点，这就是你动手的地方。 顺便说一句，如果你想知道没个程序使用了哪些系统调用，运行 strace <arguments>。

总的来说，一个用户进程是不应该也不能够直接访问内核的。它不能访问内核的内存， 也不能调用内核的函数。这是 CPU 的硬件保护机制决定的（这也是为什么叫做"保护模式"的原因）。

系统调用是这条规则的例外。所发生的事是一个进程用合适的值填充寄存器， 然后调用一条跳转到已被定义过的内核中的位置的指令（当然，这些定义过的位置是对于用户进程可读的，但是显然是不可写的）。在 Intel 架构中，这是通过 0x80 中断完成的。硬件明白一旦你跳转到这个位置， 你就不再是在处处受限的用户态中运行了，而是在无所不能的内核态中。

内核中的进程可以跳转过去的位置叫做系统调用。那儿将检查系统调用的序号， 这些序号将告诉内核用户进程需要什么样的服务。然后，通过查找系统调用表(sys_call_table) 找到内核函数的地址，调用该函数。当函数返回时， 再做一些系统检查，接着就返回用户进程（或是另一个进程，如果该进程的时间用完了）。 如果你想阅读一下这方面的源代码，它们就在文件 arch/$<$architecture$>$/kernel/entry.S 中 ENTRY(system_call)行的下面。

所以，如果我们想改变某个系统调用的运作方式，我们只需要用我们自己的函数去实现它 (通常只是加一点我们自己的代码，然后调用原函数)然后改变系统调用表 (sys_call_table)中的指针值使它指向我们的函数。因为这些模块将在以后卸载， 我们不想系统因此而不稳定，所以 cleanup_module 中恢复系统调用表是非常重要的。

这就是这样的一个模块。我们可以"监视"一个特定的用户，然后使用 printk()输出该用户打开的每个文件的消息。在结束前，我们用自己的 our_sys_open 函数替换了打开文件的系统调用。该函数检查当前进程的用户序号（uid，user's id）， 如果匹配我们监视的用户的序号，它调用 printk()输出将要打开的文件的名字。 要不然，就用同样的参数调用原始的 open()函数，真正的打开文件。

函数 init_module 改变了系统调用表中的恰当位置的值然后用一个变量保存下来。函数 cleanup_module 则使用该变量将所有东西还原。这种处理方法其实是很危险的。想象一下， 如果我们有两个这样的模块，A 和 B。A 用 A_open 替换了系统的 sys_open 函数，而 B 用 B_open。现在，

我们先把模块 A 加载， 那么原先的系统调用被 A_open 替代了，A_open 在完成工作后自身又会调用原始的 sys_open 函数 。接着，我们加载 B 模块， 它用 B_open 更改了现在的已更改为 A_open（显然它认为是原始的 sys_open 系统调用）的系统调用。

现在，如果 B 先卸载，一切正常。系统调用会还原到 A_open，而 A_open 又会调用原始的 sys_open。但是，一旦 A 先卸载，系统就会崩溃。A 的卸载会将系统调用还原到原始的 sys_open，把 B 从链中切断。 此时再卸载 B，B 会将系统调用恢复到它认为的初始状态，也就是 A_open，但 A_open 已经不在内存中了。 乍一看来，我们似乎可以通过检测系统调用是否与我们的 open 函数相同，如果不相同则什么都不做 （这样 B 就不会尝试在卸载时恢复系统调用表）。但其实这样更糟。当 A 先被卸载时，它将检测到系统 调用已被更改为 B_open，所以 A 将不会在卸载时恢复系统调用表中相应的项。此时不幸的事发生了， B_open 将仍然调用已经不存在的 A_open，这样即使你不卸载 B 模块，系统也崩溃了。

但是这种替换系统调用的方法是违背正式应用中系统的稳定和可靠原则的。所以，为了防止潜在的对系统调用表 修改带来的危害，系统调用表 sys_call_table 不再被内核导出。这意味着如果你想顺利的运行这个例子，你必须为你的 内核树打补丁来导出 sys_call_table，在 example 目录内你将找到相关的补丁和说明。正如同你想像的那样，这可不是 儿戏， 如果你的系统非常宝贵(例如这不是你的系统，或系统很难恢复)，你最好还是放弃。如果你仍然坚持，我可以 告诉你的是打补丁虽然不会有多大问题，但内核维护者他们肯定有足够的理由在 2.6 内核中不支持这种 hack。详情请参考 README。 如果你选择了 N，跳过这个例子是一个安全的选择。

Example 8-1. syscall.c

```
/*
*  syscall.c
*
*  System call "stealing" sample.
*/

/*
* Copyright (C) 2001 by Peter Jay Salzman
*/

/*
* The necessary header files
*/

/*
* Standard in kernel modules
*/
#include <linux/kernel.h>         /* We're doing kernel work */
#include <linux/module.h>          /* Specifically, a module, */
#include <linux/moduleparam.h>          /* which will have params */
#include <linux/unistd.h>         /* The list of system calls */
```

```
/*
* For the current (process) structure, we need
* this to know who the current user is.
*/
#include <linux/sched.h>
#include <asm/uaccess.h>

/*
* The system call table (a table of functions). We
* just define this as external, and the kernel will
* fill it up for us when we are insmod'ed
*
* sys_call_table is no longer exported in 2.6.x kernels.
* If you really want to try this DANGEROUS module you will
* have to apply the supplied patch against your current kernel
* and recompile it.
*/
extern void *sys_call_table[];

/*
* UID we want to spy on - will be filled from the
* command line
*/
static int uid;
module_param(uid, int, 0644);

/*
* A pointer to the original system call. The reason
* we keep this, rather than call the original function
* (sys_open), is because somebody else might have
* replaced the system call before us. Note that this
* is not 100% safe, because if another module
* replaced sys_open before us, then when we're inserted
* we'll call the function in that module - and it
* might be removed before we are.
*
* Another reason for this is that we can't get sys_open.
* It's a static variable, so it is not exported.
*/
asmlinkage int (*original_call) (const char *, int, int);
```

```
/*
* The function we'll replace sys_open (the function
* called when you call the open system call) with. To
* find the exact prototype, with the number and type
* of arguments, we find the original function first
* (it's at fs/open.c).
*
* In theory, this means that we're tied to the
* current version of the kernel. In practice, the
* system calls almost never change (it would wreck havoc
* and require programs to be recompiled, since the system
* calls are the interface between the kernel and the
* processes).
*/
asmlinkage int our_sys_open(const char *filename, int flags, int mode)
{
        int i = 0;
        char ch;

        /*
         * Check if this is the user we're spying on
         */
        if (uid == current->uid) {
                /*
                 * Report the file, if relevant
                 */
                printk("Opened file by %d: ", uid);
                do {
                        get_user(ch, filename + i);
                        i++;
                        printk("%c", ch);
                } while (ch != 0);
                printk("\n" );
        }

        /*
         * Call the original sys_open - otherwise, we lose
         * the ability to open files
         */
        return original_call(filename, flags, mode);
}
```

```
/*
* Initialize the module - replace the system call
*/
int init_module()
{
        /*
         * Warning - too late for it now, but maybe for
         * next time...
         */
        printk("I'm dangerous. I hope you did a " );
        printk("sync before you insmod'ed me.\n" );
        printk("My counterpart, cleanup_module(), is even" );
        printk("more dangerous. If\n" );
        printk("you value your file system, it will " );
        printk("be \"sync; rmmod\" \n" );
        printk("when you remove this module.\n" );

        /*
         * Keep a pointer to the original function in
         * original_call, and then replace the system call
         * in the system call table with our_sys_open
         */
        original_call = sys_call_table[__NR_open];
        sys_call_table[__NR_open] = our_sys_open;

        /*
         * To get the address of the function for system
         * call foo, go to sys_call_table[__NR_foo].
         */

        printk("Spying on UID:%d\n", uid);

        return 0;
}
```

```
/*
* Cleanup - unregister the appropriate file from /proc
*/
void cleanup_module()
{
        /*
         * Return the system call back to normal
         */
        if (sys_call_table[__NR_open] != our_sys_open) {
                printk("Somebody else also played with the " );
                printk("open system call\n" );
                printk("The system may be left in " );
                printk("an unstable state.\n" );
        }

        sys_call_table[__NR_open] = original_call;
}
```

Chapter 9. Blocking Processes

阻塞进程

Enter Sandman
当别人让你做一件你不能马上去做的事时，你会如何反映？如果你是人类的话，而且对方也是人类的话，你只会说："现在不行，我忙着在。闪开！"但是如果你是一个内核模块而且你被一个进程以同样的问题困扰， 你会有另外一个选择。你可以让该进程休眠直到你可以为它服务时。毕竟，这样的情况在内核中时时刻刻都在发生 （这就是系统让多进程在单 CPU 上同时运行的方法）。

这个内核模块就是一个这样的例子。文件（/proc/sleep)）只可以在同一时刻被一个进程打开。 如果该文件已经被打开，内核模块将调用函数 wait_event_interruptible[1]。该函数修改 task 的状态（task 是一个内核中的结构体数据结构， 其中保存着对应进程的信息和该进程正在调用的系统调用，如果有的话）为 TASK_INTERRUPTIBLE 意味着改进程将不会继续运行直到被唤醒，然后被添加到系统的进程等待队列 WaitQ 中，一个等待打开该文件的队列中。然后，该函数调用系统调度器去切换到另一个不同的 但有 CPU 运算请求的进程。

当一个进程处理完该文件并且关闭了该文件，module_close 就被调用执行了。 该函数唤醒所有在等待队列中的进程（还没有只唤醒特定进程的机制）。然后该函数返回， 那个刚刚关闭文件的进程得以继续运行。及时的，进程调度器会判定该进程执行已执行完毕， 将 CPU 转让给别的进程。被提供 CPU 使用权的那个进程就恰好从先前系统调用 module_interruptible_sleep_on[2]后的地方开始继续执行。它可以设置一个全局变量去通知别的进程该文件已被打开占用了。当别的请求该文件的进程获得 CPU 时间片时， 它们将检测该变量然后返回休眠。

更有趣的是，module_close 并不垄断唤醒等待中的请求文件的进程的权力。一个信号，像 Ctrl+c (SIGINT 也能够唤醒别的进程 [3]。 在这种情况下，我们想立即返回-EINTR 。 这对用户很重要，举个例子来说，用户可以在某个进程接受到文件前终止该进程。

还有一点值得注意。有些时候进程并不愿意休眠，它们要么立即执行它们想做的， 要么被告知任务无法进行。这样的进程在打开文件时会使用标志 O_NONBLOCK。 在别的进程被阻塞时内核应该做出的响应是返回错误代码-EAGAIN，像在本例中对该文件的请求的进程。程序 cat_noblock，在本章的源代码目录下可以找到，就能够使用标志位 O_NONBLOCK 打开文件。

Example 9-1. sleep.c

```
/*
*  sleep.c - create a /proc file, and if several processes try to open it at
*  the same time, put all but one to sleep
*/

 #include <linux/kernel.h>        /* We're doing kernel work */
 #include <linux/module.h>         /* Specifically, a module */
 #include <linux/proc_fs.h>        /* Necessary because we use proc fs */
 #include <linux/sched.h>         /* For putting processes to sleep and
```

```
                                waking them up */
#include <asm/uaccess.h>        /* for get_user and put_user */

/*
* The module's file functions
*/

/*
* Here we keep the last message received, to prove that we can process our
* input
*/
#define MESSAGE_LENGTH 80
static char Message[MESSAGE_LENGTH];

static struct proc_dir_entry *Our_Proc_File;
#define PROC_ENTRY_FILENAME "sleep"

/*
* Since we use the file operations struct, we can't use the special proc
* output provisions - we have to use a standard read function, which is this
* function
*/
static ssize_t module_output(struct file *file,         /* see include/linux/fs.h    */
                             char *buf,         /* The buffer to put data to
                                                   (in the user segment)     */
                             size_t len,        /* The length of the buffer */
                             loff_t * offset)
{
        static int finished = 0;
        int i;
        char message[MESSAGE_LENGTH + 30];

        /*
         * Return 0 to signify end of file - that we have nothing
         * more to say at this point.
         */
        if (finished) {
                finished = 0;
                return 0;
        }

        /*
         * If you don't understand this by now, you're hopeless as a kernel
         * programmer.
         */
```

```
        sprintf(message, "Last input:%s\n", Message);
        for (i = 0; i < len && message[i]; i++)
                put_user(message[i], buf + i);

        finished = 1;
        return i;                /* Return the number of bytes "read" */
}

/*
* This function receives input from the user when the user writes to the /proc
* file.
*/
static ssize_t module_input(struct file *file,          /* The file itself */
                            const char *buf,          /* The buffer with input */
                            size_t length,          /* The buffer's length */
                            loff_t * offset)
{                               /* offset to file - ignore */
        int i;

        /*
         * Put the input into Message, where module_output will later be
         * able to use it
         */
        for (i = 0; i < MESSAGE_LENGTH - 1 && i < length; i++)
                get_user(Message[i], buf + i);
        /*
         * we want a standard, zero terminated string
         */
        Message[i] = '\0';

        /*
         * We need to return the number of input characters used
         */
        return i;
}

/*
* 1 if the file is currently open by somebody
*/
int Already_Open = 0;

/*
* Queue of processes who want our file
*/
DECLARE_WAIT_QUEUE_HEAD(WaitQ);
```

```
/*
* Called when the /proc file is opened
*/
static int module_open(struct inode *inode, struct file *file)
{
        /*
         * If the file's flags include O_NONBLOCK, it means the process doesn't
         * want to wait for the file.  In this case, if the file is already
         * open, we should fail with -EAGAIN, meaning "you'll have to try
         * again", instead of blocking a process which would rather stay awake.
         */
        if ((file->f_flags & O_NONBLOCK) && Already_Open)
                return -EAGAIN;

        /*
         * This is the correct place for try_module_get(THIS_MODULE) because
         * if a process is in the loop, which is within the kernel module,
         * the kernel module must not be removed.
         */
        try_module_get(THIS_MODULE);

        /*
         * If the file is already open, wait until it isn't
         */

        while (Already_Open) {
                int i, is_sig = 0;

                /*
                 * This function puts the current process, including any system
                 * calls, such as us, to sleep.  Execution will be resumed right
                 * after the function call, either because somebody called
                 * wake_up(&WaitQ) (only module_close does that, when the file
                 * is closed) or when a signal, such as Ctrl-C, is sent
                 * to the process
                 */
                wait_event_interruptible(WaitQ, !Already_Open);

                /*
                 * If we woke up because we got a signal we're not blocking,
                 * return -EINTR (fail the system call).  This allows processes
                 * to be killed or stopped.
                 */
```

```
/*
* Emmanuel Papirakis:
*
* This is a little update to work with 2.2.*.  Signals now are contained in
* two words (64 bits) and are stored in a structure that contains an array of
* two unsigned longs.  We now have to make 2 checks in our if.
*
* Ori Pomerantz:
*
* Nobody promised me they'll never use more than 64 bits, or that this book
* won't be used for a version of Linux with a word size of 16 bits.  This code
* would work in any case.
*/
                for (i = 0; i < _NSIG_WORDS && !is_sig; i++)
                        is_sig =
                           current->pending.signal.sig[i] & ~current->
                           blocked.sig[i];

                if (is_sig) {
                        /*
                         * It's important to put module_put(THIS_MODULE) here,
                         * because for processes where the open is interrupted
                         * there will never be a corresponding close. If we
                         * don't decrement the usage count here, we will be
                         * left with a positive usage count which we'll have no
                         * way to bring down to zero, giving us an immortal
                         * module, which can only be killed by rebooting
                         * the machine.
                         */
                        module_put(THIS_MODULE);
                        return -EINTR;
                }
        }

        /*
         * If we got here, Already_Open must be zero
         */

        /*
         * Open the file
         */
        Already_Open = 1;
        return 0;               /* Allow the access */
}
```

```
/*
* Called when the /proc file is closed
*/
int module_close(struct inode *inode, struct file *file)
{
        /*
         * Set Already_Open to zero, so one of the processes in the WaitQ will
         * be able to set Already_Open back to one and to open the file. All
         * the other processes will be called when Already_Open is back to one,
         * so they'll go back to sleep.
         */
        Already_Open = 0;

        /*
         * Wake up all the processes in WaitQ, so if anybody is waiting for the
         * file, they can have it.
         */
        wake_up(&WaitQ);

        module_put(THIS_MODULE);

        return 0;               /* success */
}

/*
* This function decides whether to allow an operation (return zero) or not
* allow it (return a non-zero which indicates why it is not allowed).
*
* The operation can be one of the following values:
* 0 - Execute (run the "file" - meaningless in our case)
* 2 - Write (input to the kernel module)
* 4 - Read (output from the kernel module)
*
* This is the real function that checks file permissions. The permissions
* returned by ls -l are for reference only, and can be overridden here.
*/
static int module_permission(struct inode *inode, int op, struct nameidata *nd)
{
        /*
         * We allow everybody to read from our module, but only root (uid 0)
         * may write to it
         */
        if (op == 4 || (op == 2 && current->euid == 0))
                return 0;
```

```
        /*
         * If it's anything else, access is denied
         */
        return -EACCES;
}

/*
* Structures to register as the /proc file, with pointers to all the relevant
* functions.
*/

/*
* File operations for our proc file. This is where we place pointers to all
* the functions called when somebody tries to do something to our file. NULL
* means we don't want to deal with something.
*/
static struct file_operations File_Ops_4_Our_Proc_File = {
        .read = module_output,          /* "read" from the file */
        .write = module_input,          /* "write" to the file */
        .open = module_open,            /* called when the /proc file is opened */
        .release = module_close,          /* called when it's closed */
};

/*
* Inode operations for our proc file.  We need it so we'll have somewhere to
* specify the file operations structure we want to use, and the function we
* use for permissions. It's also possible to specify functions to be called
* for anything else which could be done to an inode (although we don't bother,
* we just put NULL).
*/

static struct inode_operations Inode_Ops_4_Our_Proc_File = {
        .permission = module_permission,          /* check for permissions */
};

/*
* Module initialization and cleanup
*/

/*
* Initialize the module - register the proc file
*/

int init_module()
{
        int rv = 0;
```

```
        Our_Proc_File = create_proc_entry(PROC_ENTRY_FILENAME, 0644, NULL);
        Our_Proc_File->owner = THIS_MODULE;
        Our_Proc_File->proc_iops = &Inode_Ops_4_Our_Proc_File;
        Our_Proc_File->proc_fops = &File_Ops_4_Our_Proc_File;
        Our_Proc_File->mode = S_IFREG | S_IRUGO | S_IWUSR;
        Our_Proc_File->uid = 0;
        Our_Proc_File->gid = 0;
        Our_Proc_File->size = 80;

        if (Our_Proc_File == NULL) {
                rv = -ENOMEM;
                remove_proc_entry(PROC_ENTRY_FILENAME, &proc_root);
                printk(KERN_INFO "Error: Could not initialize /proc/test\n" );
        }

        return rv;
}

/*
* Cleanup - unregister our file from /proc.  This could get dangerous if
* there are still processes waiting in WaitQ, because they are inside our
* open function, which will get unloaded. I'll explain how to avoid removal
* of a kernel module in such a case in chapter 10.
*/
void cleanup_module()
{
        remove_proc_entry(PROC_ENTRY_FILENAME, &proc_root);
}
```

Notes

[1] 最方便的保持某个文件被打开的方法是使用命令 tail -f 打开该文件。

[2] 这就意味着该进程仍然在内核态中， 该进程已经调用了 open 的系统调用，但系统调用却没有返回。在这段时间内该进程将不会得知别人正在使用 CPU。

[3] 这是因为我们使用的是 module_interruptible_sleep_on。我们也可以使用 module_sleep_on，但这样会导致一些十分愤怒的用户，因为他们的 Ctrl+c 将不起任何作用。

Chapter 10. Replacing Printks

替换 printk

在 the Section called 使用 X 带来的问题 in Chapter 1 中， 我说过最好不要在 X 中进行内核模块编程。在真正的内核模块开发中的确是这样。 但在实际应用中，你想在任何加载模块的 tty[1]终端中显示信息。

实现的方法是使用 current 指针，一个指向当前运行进程的指针，来获取当前任务的 tty 终端的结构体。然后，我们找到在该 tty 结构体中 用来向 tty 写入字符信息的函数的指针。通过指针我们使用该函数来向终端写入信息。

Example 10-1. print_string.c

```
/*
*  print_string.c - Send output to the tty we're running on, regardless if it's
*  through X11, telnet, etc.  We do this by printing the string to the tty
*  associated with the current task.
*/
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/sched.h>        /* For current */
#include <linux/tty.h>              /* For the tty declarations */
#include <linux/version.h>        /* For LINUX_VERSION_CODE */

MODULE_LICENSE("GPL" );
MODULE_AUTHOR(" Peter Jay Salzman" );

static void print_string(char *str)
{
        struct tty_struct *my_tty;

        /*
         * tty struct went into signal struct in 2.6.6
         */
#if ( LINUX_VERSION_CODE <= KERNEL_VERSION(2,6,5) )
        /*
         * The tty for the current task
         */
        my_tty = current->tty;
#else
```

```
        /*
         * The tty for the current task, for 2.6.6+ kernels
         */
        my_tty = current->signal->tty;
#endif

        /*
         * If my_tty is NULL, the current task has no tty you can print to
         * (ie, if it's a daemon).  If so, there's nothing we can do.
         */
        if (my_tty != NULL) {

                /*
                 * my_tty->driver is a struct which holds the tty's functions,
                 * one of which (write) is used to write strings to the tty.
                 * It can be used to take a string either from the user's or
                 * kernel's memory segment.
                 *
                 * The function's 1st parameter is the tty to write to,
                 * because the same function would normally be used for all
                 * tty's of a certain type.  The 2nd parameter controls whether
                 * the function receives a string from kernel memory (false, 0)
                 * or from user memory (true, non zero).  The 3rd parameter is
                 * a pointer to a string.  The 4th parameter is the length of
                 * the string.
                 */
                ((my_tty->driver)->write) (my_tty,         /* The tty itself */
                                           0,         /* Don't take the string
                                                  from user space        */
                                           str,         /* String                 */
                                           strlen(str));         /* Length */

                /*
                 * ttys were originally hardware devices, which (usually)
                 * strictly followed the ASCII standard.  In ASCII, to move to
                 * a new line you need two characters, a carriage return and a
                 * line feed.  On Unix, the ASCII line feed is used for both
                 * purposes - so we can't just use \n, because it wouldn't have
                 * a carriage return and the next line will start at the
                 * column right after the line feed.
                 *
                 * This is why text files are different between Unix and
                 * MS Windows.  In CP/M and derivatives, like MS-DOS and
                 * MS Windows, the ASCII standard was strictly adhered to,
                 * and therefore a newline requirs both a LF and a CR.
```

```
                */
                ((my_tty->driver)->write) (my_tty, 0, "\015\012", 2);
        }
}

static int __init print_string_init(void)
{
        print_string("The module has been inserted.  Hello world!" );
        return 0;
}

static void __exit print_string_exit(void)
{
        print_string("The module has been removed.  Farewell world!" );
}

module_init(print_string_init);
module_exit(print_string_exit);
```

Notes

[1] Teletype，原先是一种用来和 Unix 系统交互的键盘和打印机结合起来的装置。现在，它只是一个用来同 Unix 或类似的系统交流文字流 的抽象的设备，而不管它具体是显示器，X 中的 xterm，还是一个通过 telnet 的网络连接。

让你的键盘指示灯闪起来

你也许想让你的模块更直接的同外界交流，你的键盘指示灯就是一个不错的选择。它可以及时显示模块的工作状态， 吸引你的注意，并且它们不许要任何设置，使用起来也不像向终端或磁盘写入信息那么危险。

下面的这个模块代码演示了一个相当小的模块：当被加载入内核时，键盘指示灯就不停的闪烁，直到它被卸载。

Example 10-2. kbleds.c

```
/*
*  kbleds.c - Blink keyboard leds until the module is unloaded.
*/

#include <linux/module.h>
#include <linux/config.h>
#include <linux/init.h>
#include <linux/tty.h>                  /* For fg_console, MAX_NR_CONSOLES */
#include <linux/kd.h>                   /* For KDSETLED */
#include <linux/console_struct.h>       /* For vc_cons */
```

```
MODULE_DESCRIPTION("Example module illustrating the use of Keyboard LEDs.");
MODULE_AUTHOR("Daniele Paolo Scarpazza");
MODULE_LICENSE("GPL");

struct timer_list my_timer;
struct tty_driver *my_driver;
char kbledstatus = 0;

#define BLINK_DELAY   HZ/5
#define ALL_LEDS_ON   0x07
#define RESTORE_LEDS  0xFF

/*
* Function my_timer_func blinks the keyboard LEDs periodically by invoking
* command KDSETLED of ioctl() on the keyboard driver. To learn more on virtual
* terminal ioctl operations, please see file:
*     /usr/src/linux/drivers/char/vt_ioctl.c, function vt_ioctl().
*
* The argument to KDSETLED is alternatively set to 7 (thus causing the led
* mode to be set to LED_SHOW_IOCTL, and all the leds are lit) and to 0xFF
* (any value above 7 switches back the led mode to LED_SHOW_FLAGS, thus
* the LEDs reflect the actual keyboard status).  To learn more on this,
* please see file:
*     /usr/src/linux/drivers/char/keyboard.c, function setledstate().
*
*/

static void my_timer_func(unsigned long ptr)
{
        int *pstatus = (int *)ptr;

        if (*pstatus == ALL_LEDS_ON)
                *pstatus = RESTORE_LEDS;
        else
                *pstatus = ALL_LEDS_ON;

        (my_driver->ioctl) (vc_cons[fg_console].d->vc_tty, NULL, KDSETLED,
                            *pstatus);

        my_timer.expires = jiffies + BLINK_DELAY;
        add_timer(&my_timer);
}
```

```
static int __init kbleds_init(void)
{
        int i;

        printk(KERN_INFO "kbleds: loading\n");
        printk(KERN_INFO "kbleds: fgconsole is %x\n", fg_console);
        for (i = 0; i < MAX_NR_CONSOLES; i++) {
                if (!vc_cons[i].d)
                        break;
                printk(KERN_INFO "poet_atkm: console[%i/%i] #%i, tty %lx\n", i,
                       MAX_NR_CONSOLES, vc_cons[i].d->vc_num,
                       (unsigned long)vc_cons[i].d->vc_tty);
        }
        printk(KERN_INFO "kbleds: finished scanning consoles\n");

        my_driver = vc_cons[fg_console].d->vc_tty->driver;
        printk(KERN_INFO "kbleds: tty driver magic %x\n", my_driver->magic);

        /*
         * Set up the LED blink timer the first time
         */
        init_timer(&my_timer);
        my_timer.function = my_timer_func;
        my_timer.data = (unsigned long)&kbledstatus;
        my_timer.expires = jiffies + BLINK_DELAY;
        add_timer(&my_timer);

        return 0;
}

static void __exit kbleds_cleanup(void)
{
        printk(KERN_INFO "kbleds: unloading...\n");
        del_timer(&my_timer);
        (my_driver->ioctl) (vc_cons[fg_console].d->vc_tty, NULL, KDSETLED,
                            RESTORE_LEDS);
}

module_init(kbleds_init);
module_exit(kbleds_cleanup);
```

如果上面的方法都无法满足你调试的需要，你就可能需要其它的技巧了。还记得那个在 make menuconfig 时的 CONFIG_LL_DEBUG 参数吗？如果你激活该选项，你就可以获得对串口的底层操纵。如果这仍然不够爽，你还可以对 kernel/printk.c 或其它的基本的系统底层调用打补丁来使用 printascii，从而可以通过串口跟踪 内核的每步动作。如果你的架构不支持上面的例子却有一个标准的

串口，这可能应该是你首先应该考虑的了。通过网络上的 终端调试同样值得尝试。

尽管有很多关于如何调试的技巧，但我要提醒的是任何调试都会代码带来影响。加入调试代码足以导致原始代码产生 bug 的 条件的消失，所以尽可能少的加入调试代码并且确保它们不出现在成熟的代码中。

Chapter 11. Scheduling Tasks

任务调度

经常我们要定期的抽空处理一些"家务活"。如果这样的任务通过一个用户进程完成的，那么我们可以将它放到一个 crontab 文件中。如果是通过一个内核模块来完成，那么我们有两种选择。 第一种选择是使用 crontab 文件，启动一个进程，通过一个系统调用唤醒内核模块，例如打开一个文件。 这很没效率。我们通过 crontab 生成了一个新进程，读取了一段新的可执行代码进入内存， 只是为了唤醒一个已经在内存中的内核模块。

第二种选择是我们构造一个函数，然后该函数在每次时间中断发生时被调用。实现方法是我们构造一个任务，使用结构体 tq_struct，而该结构体又保存着指向该函数的指针。然后，我们用 queue_task 把该任务放在叫做 tq_timer 任务队列中。 该队列是将在下个时间中断发生时执行的任务。因为我们想要使它不停的执行，所以当该函数执行完后我们还要将它放回 tq_timer 任务队列中等待下一次时间中断。

但我们似乎忘了一点。当一个模块用 rmmod 卸载时，它会检查使用计数。 如果该计数为零，则调用 module_cleanup。然后，模块就同它的所有函数调用从内存中消失了。 此时没人去检查任务队列中是否正好还有一个等待执行的这些函数的指针。在可能是一段漫长的时间后 （当然是相对计算机而言，对于我们这点时间什么都不是，也就差不多百分之一秒吧）， 内核接收到一个时间中断，然后准备调用那个在任务队列中的函数。不幸的是，该函数已经不存在了。 大多数情况下，由于访问的内存页是空白的，你只会收到一个不愉快的消息。但是如果其它的一些代码恰好就在那里， 结果可能将会非常糟糕。同样不幸的是，我们也没有一种轻易的向任务队列注销任务的机制。

既然 cleanup_module 不能返回一个错误代码（它是一个 void 函数）， 解决之道是让它不要返回。相反，调用 sleep_on 或 module_sleep_on [1]让 rmmod 的进程休眠。在此之前，它通知被时间中断调度出任务队列的那个函数不要在返回队列。 这样，在下一个时间中断发生时，rmmod 就会被唤醒，此时我们的函数已经不在队列中， 可以很安全的卸载我们的模块了。

Example 11-1. sched.c

```
/*
*  sched.c - scheduale a function to be called on every timer interrupt.
*
*  Copyright (C) 2001 by Peter Jay Salzman
*/

/*
* The necessary header files
*/

/*
* Standard in kernel modules
*/
```

```
#include <linux/kernel.h>       /* We're doing kernel work */
#include <linux/module.h>       /* Specifically, a module */
#include <linux/proc_fs.h>      /* Necessary because we use the proc fs */
#include <linux/workqueue.h>        /* We scheduale tasks here */
#include <linux/sched.h>        /* We need to put ourselves to sleep
                                and wake up later */
#include <linux/init.h>             /* For __init and __exit */
#include <linux/interrupt.h>        /* For irqreturn_t */

struct proc_dir_entry *Our_Proc_File;
#define PROC_ENTRY_FILENAME "sched"
#define MY_WORK_QUEUE_NAME "WQsched.c"

/*
* The number of times the timer interrupt has been called so far
*/
static int TimerIntrpt = 0;

static void intrpt_routine(void *);

static int die = 0;             /* set this to 1 for shutdown */

/*
* The work queue structure for this task, from workqueue.h
*/
static struct workqueue_struct *my_workqueue;

static struct work_struct Task;
static DECLARE_WORK(Task, intrpt_routine, NULL);

/*
* This function will be called on every timer interrupt. Notice the void*
* pointer - task functions can be used for more than one purpose, each time
* getting a different parameter.
*/
static void intrpt_routine(void *irrelevant)
{
        /*
         * Increment the counter
         */
        TimerIntrpt++;
```

```
        /*
         * If cleanup wants us to die
         */
        if (die == 0)
                queue_delayed_work(my_workqueue, &Task, 100);
}

/*
* Put data into the proc fs file.
*/
ssize_t
procfile_read(char *buffer,
              char **buffer_location,
              off_t offset, int buffer_length, int *eof, void *data)
{
        int len;                /* The number of bytes actually used */

        /*
         * It's static so it will still be in memory
         * when we leave this function
         */
        static char my_buffer[80];

        static int count = 1;

        /*
         * We give all of our information in one go, so if the anybody asks us
         * if we have more information the answer should always be no.
         */
        if (offset > 0)
                return 0;

        /*
         * Fill the buffer and get its length
         */
        len = sprintf(my_buffer, "Timer called %d times so far\n", TimerIntrpt);
        count++;

        /*
         * Tell the function which called us where the buffer is
         */
        *buffer_location = my_buffer;
```

```
        /*
         * Return the length
         */
        return len;
}

/*
* Initialize the module - register the proc file
*/
int __init init_module()
{
        int rv = 0;
        /*
         * Put the task in the work_timer task queue, so it will be executed at
         * next timer interrupt
         */
        my_workqueue = create_workqueue(MY_WORK_QUEUE_NAME);
        queue_delayed_work(my_workqueue, &Task, 100);

        Our_Proc_File = create_proc_entry(PROC_ENTRY_FILENAME, 0644, NULL);
        Our_Proc_File->read_proc = procfile_read;
        Our_Proc_File->owner = THIS_MODULE;
        Our_Proc_File->mode = S_IFREG | S_IRUGO;
        Our_Proc_File->uid = 0;
        Our_Proc_File->gid = 0;
        Our_Proc_File->size = 80;

        if (Our_Proc_File == NULL) {
                rv = -ENOMEM;
                remove_proc_entry(PROC_ENTRY_FILENAME, &proc_root);
                printk(KERN_INFO "Error: Could not initialize /proc/%s\n",
                        PROC_ENTRY_FILENAME);
        }

        return rv;
}

/*
* Cleanup
*/
```

```
void __exit cleanup_module()
{
        /*
         * Unregister our /proc file
         */
        remove_proc_entry(PROC_ENTRY_FILENAME, &proc_root);
        printk(KERN_INFO "/proc/%s removed\n", PROC_ENTRY_FILENAME);

        die = 1;                /* keep intrp_routine from queueing itself */
        cancel_delayed_work(&Task);         /* no "new ones" */
        flush_workqueue(my_workqueue);         /* wait till all "old ones" finished */
        destroy_workqueue(my_workqueue);

        /*
         * Sleep until intrpt_routine is called one last time. This is
         * necessary, because otherwise we'll deallocate the memory holding
         * intrpt_routine and Task while work_timer still references them.
         * Notice that here we don't allow signals to interrupt us.
         *
         * Since WaitQ is now not NULL, this automatically tells the interrupt
         * routine it's time to die.
         */

}

/*
* some work_queue related functions
* are just available to GPL licensed Modules
*/
MODULE_LICENSE("GPL" );
```

Notes

[1] 它们实际上是一回事。

Chapter 12. Interrupt Handlers

Interrupt Handlers

除了刚结束的那章，我们目前在内核中所做的每件事都只不过是对某个请求的进程的响应， 要么是对某个特殊的文件的处理，要么是发送一个 ioctl()，要么是调用一个系统调用。 但是内核的工作不仅仅是响应某个进程的请求。还有另外一项非常重要的工作就是负责对硬件的管理。

在 CPU 和硬件之间的活动大致可分为两种。第一种是 CPU 发送指令给硬件，第二种就是硬件要返回某些信息给 CPU。 后面的那种又叫做中断，因为要知道何时同硬件对话才适宜而较难实现。硬件设备通常只有很少的缓存， 如果你不及时的读取里面的信息，这些信息就会丢失。

在 Linux 中，硬件中断被叫作 IRQ（Interrupt Requests，中断请求）[1]。有两种硬件中断，短中断和长中断。短中断占用的时间非常短，在这段时间内， 整个系统被阻塞，任何其它中断都不会处理。长中断占用的时间相对较长，在此期间，可能会有别的中断发生请求处理 （不是相同设备发出的中断）。可能的话，尽量将中断声明为长中断。

当 CPU 接收到一个中断时，它停止正在处理的一切事务（除非它在处理另一个更重要的中断， 在这种情况下它只会处理完这个重要的中断才会回来处理新产生的中断）， 将运行中的那些参数压入栈中然后调用中断处理程序。这同时意味着中断处理程序本身也有一些限制， 因为此时系统的状态并不确定。解决的办法是让中断处理程序尽快的完成它的事务，通常是从硬件读取信息和向硬件发送指令， 然后安排下一次接收信息的相关处理（这被称为"bottom half" [2])，然后返回。内核确保被安排的事务被尽快的执行。当被执行时，在内核模块中允许的操作就是被允许的。

实现的方法是调用 request_irq()函数，当接受到相应的 IRQ 时 （共有 15 种中断，在 Intel 架构平台上再加上 1 种用于串连中断控制器的中断）去调用你的中断 处理程序。该函数接收 IRQ 号，要调用的处理 IRQ 函数的名称，中断请求的类别标志位，文件 /proc/interrupts 中声明的设备的名字，和传递给中断处理程序的参数。中断请求的类别标志位可以为 SA_SHIRQ 来告诉系统你希望与其它中断处理程序共享该中断号 （这通常是由于一些设备共用相同的 IRQ 号），也可以为 SA_INTERRUPT 来告诉系统这是一个快速中断，这种情况下该函数只有在该 IRQ 空闲时才会成功返回，或者同时你又决定共享该 IQR。

然后，在中断处理程序内部，我们与硬件对话，接着使用带 tq_immediate() 和 mark_bh(BH_IMMEDIATE)的 queue_task_irq()去对 bottom half 队列进行调度。我们不能使用 2.0 版本种标准的 queue_task 的原因是中断可能就发生在别人的 queue_task[3] 中。我们需要 mark_bh 是因为早期版本的 Linux 只有一个可以存储 32 个 bottom half 的数组， 并且现在它们中的一个(BH_IMMEDIATE)已经被用来连接没有分配到队列中的入口的硬件 驱动的 bottom half。

Intel 架构中的键盘
剩余的这部分是只适用 Intel 架构的。如果你不使用 Intel 架构的平台，它们将不会工作，不要去尝试编译以下的代码。

在写这章的事例代码时，我遇到了一些困难。一方面，我需要一个可以得到实际有意义结果的， 能在各种平台上工作的例子。另一方面，内核中已经包括了各种设备驱动，并且这些驱动将无法和我的例子共

存。 我找到的解决办法是为键盘中断写点东西，当然首先禁用普通的键盘中断。因为该中断在内核中定义为一个静态连接的符号 （见 drivers/char/keyboard.c)），我们没有办法恢复。所以在 insmod 前，如果你爱惜你的机器，新打开一个终端运行 sleep 120 ; reboot。

该代码将自己绑定在 IRQ 1， 也就是 Intel 架构中键盘的 IRQ。然后，当接收到一个键盘中断请求时，它读取键盘的状态（那就是 inb(0x64)的目的）和扫描码，也就是键盘返回的键值。然后，一旦内核认为这是符合条件的，它运行 got_char 去给出操作的键（扫描码的头 7 个位）和是按下键（扫描码的第 8 位为 0） 还是弹起键（扫描码的第 8 位为 1)。

Example 12-1. intrpt.c

```
/*
*  intrpt.c - An interrupt handler.
*
*  Copyright (C) 2001 by Peter Jay Salzman
*/

/*
* The necessary header files
*/

/*
* Standard in kernel modules
*/
#include <linux/kernel.h>         /* We're doing kernel work */
#include <linux/module.h>          /* Specifically, a module */
#include <linux/sched.h>
#include <linux/workqueue.h>
#include <linux/interrupt.h>         /* We want an interrupt */
#include <asm/io.h>

#define MY_WORK_QUEUE_NAME "WQsched.c"

static struct workqueue_struct *my_workqueue;

/*
* This will get called by the kernel as soon as it's safe
* to do everything normally allowed by kernel modules.
*/
static void got_char(void *scancode)
{
        printk("Scan Code %x %s.\n",
               (int)*((char *)scancode) & 0x7F,
               *((char *)scancode) & 0x80 ? "Released" : " Pressed" );
}
```

```
/*
* This function services keyboard interrupts. It reads the relevant
* information from the keyboard and then puts the non time critical
* part into the work queue. This will be run when the kernel considers it safe.
*/
irqreturn_t irq_handler(int irq, void *dev_id, struct pt_regs *regs)
{
        /*
         * This variables are static because they need to be
         * accessible (through pointers) to the bottom half routine.
         */
        static int initialised = 0;
        static unsigned char scancode;
        static struct work_struct task;
        unsigned char status;

        /*
         * Read keyboard status
         */
        status = inb(0x64);
        scancode = inb(0x60);

        if (initialised == 0) {
                INIT_WORK(&task, got_char, &scancode);
                initialised = 1;
        } else {
                PREPARE_WORK(&task, got_char, &scancode);
        }

        queue_work(my_workqueue, &task);

        return IRQ_HANDLED;
}

/*
* Initialize the module - register the IRQ handler
*/
int init_module()
{
        my_workqueue = create_workqueue(MY_WORK_QUEUE_NAME);
```

```
        /*
         * Since the keyboard handler won't co-exist with another handler,
         * such as us, we have to disable it (free its IRQ) before we do
         * anything.  Since we don't know where it is, there's no way to
         * reinstate it later - so the computer will have to be rebooted
         * when we're done.
         */
        free_irq(1, NULL);

        /*
         * Request IRQ 1, the keyboard IRQ, to go to our irq_handler.
         * SA_SHIRQ means we're willing to have othe handlers on this IRQ.
         * SA_INTERRUPT can be used to make the handler into a fast interrupt.
         */
        return request_irq(1,         /* The number of the keyboard IRQ on PCs */
                           irq_handler,         /* our handler */
                           SA_SHIRQ, "test_keyboard_irq_handler",
                           (void *)(irq_handler));
}

/*
* Cleanup
*/
void cleanup_module()
{
        /*
         * This is only here for completeness. It's totally irrelevant, since
         * we don't have a way to restore the normal keyboard interrupt so the
         * computer is completely useless and has to be rebooted.
         */
        free_irq(1, NULL);
}

/*
* some work_queue related functions are just available to GPL licensed Modules
*/
MODULE_LICENSE("GPL" );
```

Notes

[1] 这是 Linux 起源的 Intel 架构中的标准的起名方法。

[2] 这里是译者给出的关于“bottom half”的一点解释，来源是 google 上搜索到的英文资料：

“底部”，“bottom half”常在涉及中断的设备驱动中提到。

当内核接收到一个中断请求，对应的设备驱动被调用。因为在这段时间内无法处理别的任何事务， 让中断处理尽快的完成并重新让内核返回正常的工作状态是非常重要的。就是因为这个设计思想， 驱动的"顶部"和"底部"的概念被提出："顶部"是被内核调用时最先被执行的部分， 快速的完成一些尽量少的却是必需的工作（像对硬件或其它资源的独享访问这种必须立刻执行的操作)， 然后做一些设置让"底部"去完成那些要求时间相对比较宽裕的，剩下的工作。

"底部"什么时候如何运作是内核的设计问题。你也许会听到"底部"的设计已经在最近的内核中被废除了。这种说法不是很确切，在新内核中其实你可以去选择怎样去执行：像软中断或任务，就像它们以前那样，还是加入任务队列，更像启动一个用户进程。

[3] queue_task_irq 被一个全局的锁（有锁定作用的变量） 保护着，在版本 2.2 中，并没有 queue_task_irq 而且 queue_task 也是被一个锁保护的。

Chapter 13. Symmetric Multi Processing

对称多线程处理

提高性能的最简单也是最便宜的方法是给你的主板加第二个 CPU（如果你的主板支持的话）。 这可以通过让不同的 CPU 完成不同的工作（非对称多线程处理）或是相同的工作（对称多线程处理）。 实现高效率的非对称的多线程处理需要特殊硬件相关的知识，而对于 Linux 这样通用操作系统这是不可能的。 相对而言，对称多线程处理是较容易实现的。

我这里所说的相对容易，老实说，还是不容易。在一个对称多线程处理的环境中， 多个 CPU 共享内存，导致的结果是其中一个 CPU 运行的代码会对别的 CPU 也产生影响。 你不能再确定你代码中第一行中设置的变量在接下来的那行代码中还是那个设置值； 其它的 CPU 可能会趁你不注意已经把它修改了。显然，如果是这样的话，是无法进行任何编程的。

对于进程层面上的编程这通常不是个问题，因为一个进程通常同一时间只在一个 CPU 上运行 [1]。 但是，对于内核，就可以被在不同的 CPU 上的同时运行的不同的进程使用。

在内核版本 2.0.x 中，这还不算作什么问题，因为整个内核是一个 spinlock [2]，这就意味着一旦某个 CPU 进入内核态，别的 CPU 将不允许进入内核态。这使 Linux 的 SMP 实现很安全 [3]，但缺乏效率。

在内核版本 2.2.x 以后，多 CPU 已经允许同时进入内核态。内核模块的作者应该意识到这一点。

Notes
[1] 存在例外，就是线程化的进程，这样的进程可以在多个 CPU 上同时运行。

[2] 抱歉，我没有找到合适的词语来表达这个单词。这是内核中的一种机制，可以对内核中的关键数据结构进行锁定保护， 防止其被破坏。

[3] 意味着这样的 SMP 机制使用起来很安全。

Chapter 14. Common Pitfalls

注意

在我让你们进入内核模块的世界之前，我需要提醒你们下面的一些注意。如果我没警告到你们但是的确发生了， 那么你将问题报告我，我将全额退还你的书款。

使用标准库文件:
你无法这样做。在内核模块中，你只能使用内核提供的函数，也就是你在 /proc/kallsyms 能查到的那些。

禁用中断:
你如果这样做了但只是一瞬间，没问题，当我没提这事。但是事后你没有恢复它们， 你就只能摁电源键来重启你僵死的系统了。

尝试一些非常危险的东西:
这也许不应该由我来说，但是以防万一，我还是提出来吧！

Appendix A. Changes: 2.0 To 2.2
从 2.0 到 2.2 的变化
从 2.0 到 2.2 的变化
我对内核的了解并不很完全所以我也无法写出所有的变化。在修改代码 （更确切的说，是采用 Emmanuel Papirakis 的修改）时，我遇到了以下的这些修改。 我将它们都列出来以方便模块编写者们，特别是学习该档案先前版本并熟悉我提到的这些技巧 （但已经更换到新版本的）的那些人。

更多的这方面的参考资料在 Richard Gooch's 的站点上。

asm/uaccess.h
如果你要使用 put_user 或 get_user 你就需要 #include 它。

get_user
在 2.2 版本中，get_user 同时接收用户内存的指针和用来 设置信息的内核内存中变量的内存指针。变化的原因是因为当我们读取的变量是二或四个字节长的时候， get_user 也可以读取二或四个字节长的变量。

file_operations
改结构体现在有了一个可以在 open 和 close 之间进行的刷新操作函数。

close in file_operations
2.2 版本中，close 返回整形值，所以可以检测是否失败。

read,write in file_operations
这些函数的头文件改变了。它们现在返回 ssize_t 而不是整形值， 且它们的参数表也变了。inode 不再是一个参数，文件中的偏移量也一样。

proc_register_dynamic
该函数已经不复存在。你应该使用用 0 作为 inode 参数的 proc_register 函数来替代它。

Signals
在 task 结构体中的 signals 不再是一个 32 位整形变量，而是一个为 _NSIG_WORDS 整形的数组。

queue_task_irq
即使你想在中断处理内部调度一个任务，你也应该使用 queue_task 而不是 queue_task_irq。

Module Parameters
你不必在将模块参数声明为全局变量。在 2.2 中，使用 MODULE_PARM 去声明模块参数。这是一个进步， 这样就允许模块接受以数字开头的参数名而不会被弄糊涂。

Symmetrical Multi-Processing
内核本省已不再是一个 spinlock，意味着你的模块也应该考虑 SMP 的问题。

Appendix B. Where To Go From Here

为什么这样写？

我其实可以给这本书再加入几章，例如如何为实现新的文件系统加上一章，或是添加一个新的协议栈（如果有这样的必要的话， 想找到 Linux 不支持的网络协议已经是非常的困难的了）。我还可以解释一下我们尚未接触到的内核实现机制，像系统的引导自举， 或磁盘存储。

但是，我决定否。我写本书的目的是提供基本的，入门的对神秘的内核模块编程的认识和这方面的常用技巧。 对于那些非常热衷与内核编程的人，我推荐 Juan-Mariano de Goyeneche 的 内核资源列表 。同样，就同 Linus 本人说的那样，学习内核最好的方法是自己阅读内核源代码。

如果你对更多的短小的示例内核模块感兴趣，我向你推荐 Phrack magazine 这本杂志。 即使你不关心安全问题，作为一个程序员你还是应该时时考虑这个问题的。这些内核模块代码都很短，不需要费多大劲就能读懂。

我希望我满足了你希望成为一个更优秀的程序员的要求，至少在学习技术的过程中体会到了乐趣。 如果你真的写了一些非常有用的模块，我希望你使用 GPL 许可证发布你的模块，这样我也就可以使用它们了。