Linux 内核编程

著者：Ori Pomerantz

翻译：徐辉

2000 **年** 8 **月** 19 **日**

译者前言

这是我的第一次尝试，在此之前我还没有接触过 Linux，所以翻译得很粗糙，有的地方我自己也不明白，只好照着翻下来。而且急急匆匆，毛毛草草，一定有许多错误或不当之处。我一向就是这么毛草的啦，总是给我的组织丢脸。☺所以如果你发现了有什么错误或者解释不清的地方，希望能够指正，敬请把您的金玉之言发到我的信箱里。

本人此举旨在结识天下 Linux 英雄。本人徐辉（号：水光月影，真命天子）现在北大方正研究院读研，主要研究方向是信息安全、数据加密和 Linux 的安全性。由于我们的工作在方正尚属开创，所以希望能够结识最多的 Linux、网络安全方面的高手。如果您有什么项目需要合作，或者有什么好的提议，或者有关于安全方面的需求，或者有比较好的资料，敬请与我们联系。本人将感激不尽。☺ //bow

本书英文下载版可在 http://metalab.unc.edu/ldp 找到。印刷版请见书后的说明。

最后必须声明：本书翻译完全是个人行为，我只代表我个人。本资料为内部交流使用，未经作者及译者许可，任何单位和个人不得将本资料用作商业用途。如经发现，本人有权力追究法律责任。

译者 email:xu_hui@icst.pku.edu.cn

2000 年 8 月 19 日　于北大燕园

目　录

1．Hello, world

当第一个穴居的原始人程序员在墙上凿出第一个“洞穴计算机”的程序时，那是一个打印出用羚羊角上的图案表示的“Hello world”的程序。罗马编程教科书上是以“Salut, Mundi”的程序开始的。我不知道如果人们打破这个传统后会有什么后果，但我认为还是不要去发现这个后果比较安全。

一个内核模块至少包括两个函数：init_module，在这个模块插入内核时调用；cleanup_module，在模块被移出时调用。典型情况下，init_module 为内核中的某些东西注册一个句柄，或者把内核中的程序提换成它自己的代码（通常是进行一些工作以后再调用原来工作的代码）。Clean_module 模块要求撤销 init_module 进行的所有处理工作，使得模块可以被安全的卸载。

Exhello.c

```
/* hello.c
 * Copyright (C) 1998 by Ori Pomerantz
 *
 * "Hello, world" - the kernel module version.
 */

/* The necessary header files */

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



/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif



/* Initialize the module */
int init_module()
{
   printk("Hello, world - this is the kernel speaking\n");

   /* If we return a non zero value, it means that
    * init_module failed and the kernel module
    * can't be loaded */
   return 0;
```

```
}


/* Cleanup - undid whatever init_module did */
void cleanup_module()
{
  printk("Short is the life of a kernel module\n");
}
```

1．1 内核模块的编译文件

一个内核模块不是一个可以独立执行的文件，而是需要在运行时刻连接入内核的目标文件。所以，它们需要用-c 选项进行编译。而且，所有的内核模块都必须包含特定的标志：

- __KERNEL__——这个标志告诉头文件此代码将在内核模块中运行，而不是作为用户进程。
- MODULE——这个标志告诉头文件要给出适当的内核模块的定义。
- LINUX——从技术上讲，这个标志不是必要的。但是，如果你希望写一个比较正规的内核模块，在多个操作系统上编译，这个标志将会使你感到方便。它可以允许你在独立于操作系统的部分进行常规的编译。

还有其它的一些可被选择包含标志，取决于编译模块是的选项。如果你不能明确内核怎样被编译，可以在 in/usr/include/linux/config.h 中查到。

- __SMP__——对称多线程。在内核被编译成支持对称多线程（尽管在一台处理机上运行）是必须定义。如果是这样，还需要做一些别的事情（参见第 12 章）。
- CONFIG_MODVERSIONS——如果 CONFIG_MODVERSIONS 被激活，你需要在编译是定义它并且包含文件/usr/include/linux/modversions.h。这可以有代码自动完成。

ex Makefile

```
# Makefile for a basic kernel module

CC=gcc
MODCFLAGS := -Wall -DMODULE -D__KERNEL__ -DLINUX

hello.o: hello.c /usr/include/linux/version.h
                $(CC) $(MODCFLAGS) -c hello.c
                echo insmod hello.o to turn it on
                echo rmmod hello to turn if off
                echo
                echo X and kernel programming do not mix.
                echo Do the insmod and rmmod from outside
```

所以，并不是剩下的事情就是 root（你没有把它编译成 root，而是在边缘（注 1.1）。对吗？），然后就在你的核心内容里插入或移出 hello。当你这样做的时候，要注意到你的新模块在/proc/modules 里。

而且，编译文件不推荐从 X 下插入的原因是内核有一条需要用 printk 打印的消息，它把它送给了控制台。如果你不使用 X，它就送到了你使用的虚拟终端（你用 Alt-F<n>选择的哪个）并且你可以看到。相反的，如果你使用了 X，就有两种可能性。如果用 xterm –C 打开了一个控制台，输出将被送到哪里。如果没有，输出将被送到虚拟终端 7——被 X“覆盖”的那个。

如果你的内核变得不稳定，你可以在没有 X 的情况下得到调试消息。在 X 外，printk 可以直接从内核中输出到控制台。而如果在 X 里，printk 输出到一个用户态的进程（xterm –C）。当进程接收到 CPU 时间，它会将其送到 X 服务器进程。然后，当 X 服务器进程接收到 CPU 时间，它将会显示，但是一个不稳定的内核意味着系统将会崩溃或重起，所以你不希望显示错误的消息，然后可能被解释给你什么发生了错误，但是超出了正确的时间。

1.2 多文件内核模块

有些时候在几个源文件之间分出一个内核模块是很有意义的。在这种情况下，你需要做下面的事情：

1. 在除了一个以外的所有源文件中，增加一行#define __NO_VERSION__。这是很重要的，因为 module.h 一般包括 kernel_version 的定义，这是一个全局变量，包含模块编译的内核版本。如果你需要 version.h，你需要把自己把它包含进去，因为如果有__NO_VERSION__的话 module.h 不会自动包含。
2. 象通常一样编译源文件。
3. 把所有目标文件联编成一个。在 X86 下，用 ld –m elf_i386 –r –o <name of module>.o <1st source file>

这里给出一个这样的内核模块的例子。

ex **start.c**

```
/* start.c
 * Copyright (C) 1999 by Ori Pomerantz
 *
 * "Hello, world" - the kernel module version.
 * This file includes just the start routine
 */

/* The necessary header files */

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



/* Deal with CONFIG_MODVERSIONS */
```

```
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif



/* Initialize the module */
int init_module()
{
   printk("Hello, world - this is the kernel speaking\n");

   /* If we return a non zero value, it means that
     * init_module failed and the kernel module
     * can't be loaded */
   return 0;
}
```

ex **stop.c**

```
/* stop.c
 * Copyright (C) 1999 by Ori Pomerantz
 *
 * "Hello, world" - the kernel module version. This
 * file includes just the stop routine.
 */

/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>      /* We're doing kernel work */

#define __NO_VERSION__           /* This isn't "the" file
                                 * of the kernel module */
#include <linux/module.h>      /* Specifically, a module */

#include <linux/version.h>      /* Not included by
                                   * module.h because
                                   * of the __NO_VERSION__ */



/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
```

```
#include <linux/modversions.h>
#endif



/* Cleanup - undid whatever init_module did */
void cleanup_module()
{
   printk("Short is the life of a kernel module\n");
}
```

ex **Makefile**

```
# Makefile for a multifile kernel module

CC=gcc
MODCFLAGS := -Wall -DMODULE -D__KERNEL__ -DLINUX

hello.o:    start.o stop.o
            ld -m elf_i386 -r -o hello.o start.o stop.o

start.o:    start.c /usr/include/linux/version.h
            $(CC) $(MODCFLAGS) -c start.c

stop.o:          stop.c /usr/include/linux/version.h
            $(CC) $(MODCFLAGS) -c stop.c
```

2．字符设备文件

那么，现在我们是原始级的内核程序员，我们知道如何写不做任何事情的内核模块。我们为自己而骄傲并且高昂起头来。但是不知何故我们感觉到缺了什么东西。患有精神紧张症的模块不是那么有意义。

内核模块同进程对话有两种主要途径。一种是通过设备文件（比如/dev 目录中的文件），另一种是使用 proc 文件系统。我们把一些东西写入内核的一个主要原因就是支持一些硬件设备，所以我们从设备文件开始。

设备文件的最初目的是允许进程同内核中的设备驱动通信，并且通过它们和物理设备通信（modem，终端，等等）。这种方法的实现如下：

每个设备驱动都对应着一定类型的硬件设备，并且被赋予一个主码。设备驱动的列表和它们的主码可以在 in/proc/devices 中找到。每个设备驱动管理下的物理设备也被赋予一个从码。无论这些设备是否真的安装，在/dev 目录中都将有一个文件，称作设备文件，对应着每一个设备。

例如，如果你进行 ls –l /dev/hd[ab] *操作，你将看见可能联结到某台机器上的所有的 IDE 硬盘分区。注意它们都使用了同一个主码，3，但是从码却互不相同。（声明：这是在 PC 结构上的情况，我不知道在其他结构上运行的 linux 是否如此。）

在系统安装时，所有设备文件在 mknod 命令下被创建。它们必须创建在/dev 目录下没有技术上的原因，只是一种使用上的便利。如果是为测试目的而创建的设备文件，比如我们这里的练习，可能放在你编译内核模块的的目录下更加合适。

设备可以被分成两类：字符设备和块设备。它们的区别是块设备有一个用于请求的缓冲区，所以它们可以选择用什么样的顺序来响应它们。这对于存储设备是非常重要的，读取相邻的扇区比互相远离的分区速度会快得多。另一个区别是块设备只能按块（块大小对应不同设备而变化）接受输入和返回输出，而字符设备却按照它们能接受的最少字节块来接受输入。大部分设备是字符设备，因为它们不需要这种类型的缓冲。你可以通过观看 ls -l 命令的输出中的第一个字符而知道一个设备文件是块设备还是字符设备。如果是 b 就是块设备，如果是 c 就是字符设备。

这个模块可以被分成两部分：模块部分和设备及设备驱动部分。Init_module 函数调用 module_register_chrdev 在内核得块设备表里增加设备驱动。同时返回该驱动所使用的主码。Cleanup_module 函数撤销设备的注册。

这些操作（注册和注销）是这两个函数的主要功能。内核中的函数不是象进程一样自发运行的，而是通过系统调用，或硬件中断或者内核中的其它部分（只要是调用具体的函数）被进程调用的。所以，当你向内和中增加代码时，你应该把它注册为具体某种事件的句柄，而当你把它删除的时候，你需要注销这个句柄。

设备驱动完全由四个设备_<action〉函数构成，它们在希望通过有主码的设备文件实现一些操作时被调用。内核调用它们的途径是通过 file_operation 结构 Fops。此结构在设备被注册是创建，它包含指向这四个函数的指针。

另一点我们需要记住的是，我们不能允许管理员随心所欲的删除内核模块。这是因为如果设备文件是被进程打开的，那么我们删除内核模块的时候，要使用这些文件就会导致访问正常的函数（读/写）所在的内存位置。如果幸运，那里不会有其他代码被装载，我们将得到一个恶性的错误信息。如果不行，另一个内核模块会被装载到同一个位置，这将意味着会跳入内核中另一个程序的中间，结果将是不可预料的恶劣。

通常你不希望一个函数做什么事情的时候，会从那个函数返回一个错误码（一个负数）。但这在 cleanup_module 中是不可能的，因为它是一个 void 型的函数。一旦 cleanup_module

被调用，这个模块就死掉了。然而有一个计数器记录着有多少个内核模块在使用这个模块，这个计数器称为索引计数器（/proc/modules **中没行的最后一个数字）。如果这个数字不是** 0**，删除就会失败。模块的索引计数器包含在变量** mod_use_count_**中。有定义好的处理这个变量的宏（**MOD_INC_USE_COUNT **和** MOD_DEC_USE_COUNT**），所以我们一般使用宏而不是直接使用变量** mod_use_count_**，这样在以后实现变化的时候会带来安全性。**

ex chardev.c

```
/* chardev.c
 * Copyright (C) 1998-1999 by Ori Pomerantz
 *
 * Create a character device (read only)
 */

/* The necessary header files */

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

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* For character devices */
#include <linux/fs.h>            /* The character device
                                    * definitions are here */
#include <linux/wrapper.h>    /* A wrapper which does
                                    * next to nothing at
                                    * at present, but may
                                    * help for compatibility
                                    * with future versions
                                    * of Linux */


/* In 2.2.3 /usr/include/linux/version.h includes
 * a macro for this, but 2.0.35 doesn't - so I add
 * it here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif
```

```
/* Conditional compilation. LINUX_VERSION_CODE is
  * the code (as per KERNEL_VERSION) of this version. */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>    /* for put_user */
#endif



#define SUCCESS 0


/* Device Declarations ***************************** */

/* The name for our device, as it will appear
  * in /proc/devices */
#define DEVICE_NAME "char_dev"


/* The maximum length of the message from the device */
#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 function is called whenever a process
  * attempts to open the device file */
static int device_open(struct inode *inode,
                       struct file *file)
{
   static int counter = 0;

#ifdef DEBUG
   printk ("device_open(%p,%p)\n", inode, file);
```

+

```
#endif

   /* This is how you get the minor device number in
    * case you have more than one physical device using
    * the driver. */
   printk("Device: %d.%d\n",
          inode->i_rdev >> 8, inode->i_rdev & 0xFF);

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

   /* If this was a process, we would have had to
   * be more careful here.
   *
   *In the case of processes, the danger would be
   *that one process might have check Device_Open
   *and then be replaced by the schedualer by another
   *process which runs this function. Then, when
   *the first process was back on the CPU, it would assume
   *the device is still not open.
   * However, Linux guarantees that a process won't
   * be replaced while it is running in kernel context.
    *
   * In the case of SMP, one CPU might increment
   *Device_Open while another CPU is here, right after the check.
   *However, in version 2.0 of the kernel this is not a problem
   *because there's a lock to guarantee only one CPU will
   *be kernel module at the same time.
   *This is bad in    terms of    performance, so version 2.2 changed it.
   *Unfortunately, I don't have access to an SMP box
   *to check how it works with SMP.
    */

   Device_Open++;

   /* Initialize the message. */
   sprintf(Message,
      "If I told you once, I told you %d times - %s",
      counter++,
      "Hello, world\n");
   /* The only reason we're allowed to do this sprintf
    * is because the maximum length of the message
```

+−

```
     * (assuming 32 bit integers - up to 10 digits
     * with the minus sign) is less than BUF_LEN, which
     * is 80. BE CAREFUL NOT TO OVERFLOW BUFFERS,
     * ESPECIALLY IN THE KERNEL!!!
     */

    Message_Ptr = Message;

    /* Make sure that the module isn't removed while
     * the file is open by incrementing the usage count
     * (the number of opened references to the module, if
     * it's not zero rmmod will fail)
     */
    MOD_INC_USE_COUNT;

    return SUCCESS;
}


/* This function is called when a process closes the
  * device file. It doesn't have a return value in
  * version 2.0.x because it can't fail (you must ALWAYS
  * be able to close a device). In version 2.2.x it is
  * allowed to fail - but we won't let it.
  */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static int device_release(struct inode *inode,
                          struct file *file)
#else
static void device_release(struct inode *inode,
                           struct file *file)
#endif
{
#ifdef DEBUG
    printk ("device_release(%p,%p)\n", inode, file);
#endif

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

    /* Decrement the usage count, otherwise once you
     * opened the file you'll never get rid of the module.
     */
    MOD_DEC_USE_COUNT;
```

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
   return 0;
#endif
}


/* This function is called whenever a process which
  * have already opened the device file attempts to
  * read from it. */


#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_read(struct file *file,
        char *buffer,        /* The buffer to fill with data */
        size_t length,       /* The length of the buffer */
        loff_t *offset)    /* Our offset in the file */
#else
static int device_read(struct inode *inode,
                                    struct file *file,
        char *buffer,      /* The buffer to fill with
                    * the data */
        int length)          /* The length of the buffer
                                 * (mustn't write beyond that!) */
#endif
{
   /* Number of bytes actually written to the buffer */
   int bytes_read = 0;

   /* 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 - unsupported in this example. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_write(struct file *file,
        const char *buffer,        /* The buffer */
        size_t length,      /* The length of the buffer */
        loff_t *offset)    /* Our offset in the file */
#else
static int device_write(struct inode *inode,
                                    struct file *file,
                                    const char *buffer,
                                    int length)
#endif
{
    return -EINVAL;
}




/* Module Declarations ***************************** */

/* The major device number for the device. This is
  * global (well, static, which in this context is global
  * within this file) because it has to be accessible
  * both for registration and for release. */
```

```
static int Major;

/* 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 = {
   NULL,      /* seek */
   device_read,
   device_write,
   NULL,      /* readdir */
   NULL,      /* select */
   NULL,      /* ioctl */
   NULL,      /* mmap */
   device_open,
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
   NULL,      /* flush */
#endif
   device_release    /* a.k.a. close */
};


/* Initialize the module - Register the character device */
int init_module()
{
   /* Register the character device (atleast try) */
   Major = module_register_chrdev(0,
                                          DEVICE_NAME,
                                          &Fops);

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

   printk ("%s The major device number is %d.\n",
               "Registeration is a success.",
               Major);
```

```
    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 <name> c %d <minor>\n", Major);
    printk ("You can try different minor numbers %s",
                "and see what happens.\n");

    return 0;
}


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

    /* Unregister the device */
    ret = module_unregister_chrdev(Major, DEVICE_NAME);

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

2．1 多内核版本源文件

系统调用是内核出示给进程的主要接口，在不同版本中一般是相同的。可能会增加新的系统，但是旧的系统的行为是不变的。向后兼容是必要的——新的内核版本不能打破正常的进程规律。在大多数情况下，设备文件是不变的。然而，内核中的内部接口是可以在不同版本间改变的。

Linux **内核的版本分为稳定版（**n.<**偶数**>.m**）和发展版（**n.<**奇数**>.m**）。发展版包含了所有新奇的思想，包括那些在下一版中被认为是错的，或者被重新实现的。所以，你不能相信在那些版本中这些接口是保持不变的（这就是为什么我在本书中不厌其烦的支持不同接口。这是很大量的工作但是马上就会过时）。但是在稳定版中我们就可以认为接口是相同的，即使在修正版中（数字** m **所指的）。**

MPG **版本包括了对内核** 2.0.x **和** 2.2.x **的支持。这两种内核仍有不同之处，所以编译时要取决于内核版本而决定。方法是使用宏** LINUX_VERSION_CODE**。在** a.b.c **版中，这个宏的值是** $2^{16}a+2^{8}b+c$**。如果希望得到具体内核版本号，我们可以使用宏** KERNEL_VERSION**。在** 2.0.35 **版中没有定义这个宏，在需要时我们可以自己定义。**

3．/proc 文件系统

在 Linux 中有一个另外的机制来使内核及内核模块发送信息给进程——/proc 文件系统。/proc 文件系统最初是设计使得容易得到进程的信息（从名字可以看出），现在却被任意一块有内容需要报告的内核使用，比如拥有模块列表的/proc/modules 和拥有内存使用统计信息的/proc/meminfo。

使用 proc 文件系统的方法很象使用设备驱动——你创建一个数据结构，使之包含/proc 文件需要的全部信息，包括所有函数的句柄（在我们的例子里只有一个，在试图读取/proc 文件时调用）。然后，用 init_module 注册这个结构，用 cleanup_module 注销。

我们使用 proc_register_dynamic（注 3.1）的原因是我们不希望决定以后在文件中使用的索引节点数，而是让内核来决定它，为了防止冲突。标准的文件系统是在磁盘上而不是在内存（/proc 的位置在内存），在这种情况下节点数是一个指向文件的索引节点所在磁盘地址的指针。这个索引节点包含了文件的有关信息比如文件的访问权限以及指向磁盘地址的指真或者文件数据的位置。

因为在文件打开或关闭时我们没有调用，所以在模块里无处可放宏 MOD_INC_USE_COUNT 和 MOD_DEC_USE_COUNT，而且如果文件被打开了或者模块被删除了，就没有办法来避免这个结果。下一章我们将会看到一个更困难的处理/proc 的方法，但是也更加灵活，也能够解决这个问题。

ex **procfs.c**

```
/* procfs.c -    create a "file" in /proc
 * Copyright (C) 1998-1999 by Ori Pomerantz
 */


/* The necessary header files */

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

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif


/* Necessary because we use the proc fs */
#include <linux/proc_fs.h>



/* In 2.2.3 /usr/include/linux/version.h includes a
```

```
  * macro for this, but 2.0.35 doesn't - so I add it
  * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



/* 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. Zero (for future use?).


    Usage and Return Value
    ======================
    If you use your own buffer, like I do, put its
    location in the second argument and return the
    number of bytes used in the buffer.

    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.
 */
int procfile_read(char *buffer,
                  char **buffer_location,
                  off_t offset,
                  int buffer_length,
                  int zero)
{
  int len;    /* The number of bytes actually used */

  /* This is 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
   * 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)
    return 0;

  /* Fill the buffer and get its length */
  len = sprintf(my_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++;

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

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


struct proc_dir_entry Our_Proc_File =
   {
      0, /* Inode number - ignore, it will be filled by
            * proc_register[_dynamic] */
      4, /* Length of the file name */
      "test", /* The file name */
      S_IFREG | S_IRUGO, /* File mode - this is a regular
                          * file which can be read by its
                          * owner, its group, and everybody
                          * else */
      1,     /* Number of links (directories where the
              * file is referenced) */
      0, 0,    /* The uid and gid for the file - we give it
                * to root */
      80, /* The size of the file reported by ls. */
      NULL, /* functions which can be done on the inode
             * (linking, removing, etc.) - we don't
             * support any. */
      procfile_read, /* The read function for this file,
                      * the function called when somebody
                      * tries to read something from it. */
      NULL /* We could have here a function to fill the
            * file's inode, to enable us to play with
            * permissions, ownership, etc. */
   };




/* Initialize the module - register the proc file */
int init_module()
{
   /* Success if proc_register[_dynamic] is a success,
    * failure otherwise. */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
   /* In version 2.2, proc_register assign a dynamic
    * inode number automatically if it is zero in the
    * structure , so there's no more need for
```

```
     * proc_register_dynamic
     */
    return proc_register(&proc_root, &Our_Proc_File);
#else
    return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif

    /* proc_root is the root directory for the proc
     * fs (/proc). This is where we want our file to be
     * located.
     */
}


/* Cleanup - unregister our file from /proc */
void cleanup_module()
{
    proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

4．使用/proc 进行输入

现在我们已经有了两种方法从内核模块中产生输出：注册一个设备驱动并且 mknod 一个设备文件，或者创建一个/proc 文件。这可以使内核告诉我们任何信息。现在的问题是我们没有办法回答给内核。我们象内核输入的第一种方法是写给/proc 文件。

因为 proc 文件系统主要是为满足内核向进程报告其状态的，没有为输入留出特别的规定。数据结构 proc_dir_entry 没有包含一个指向某个输入函数的指针，就象指向输出函数那样。如果我们要向一个/proc 文件写入，我们需要使用标准文件系统机制。

在 Linux 里有一个文件系统注册的标准机制。每个文件系统都有自己的函数来处理索引节点和文件操作，所以就有一个特殊的机构来存放指向所有函数的指针，struct inode_operations，它有一个指向 struct file_operations 的指针。在/proc 里，无论何时我们注册一个新文件，我们就被允许指定用 inod_operations 访问哪个结构。这就是我们要用的机制，一个 inode_operations，包括一个指向 file_operations 的指针，file_operations 里包含我们的 module_input 和 module_output 函数。

必须指出标准的读写角色在内核中被倒置了，读函数用来输出，而写函数用来输入。这是因为读和写是在用户的观点看，如果一个进程从内核中读取一些内容，那么内核就必须输出处理。而进程要写入内核，内核就要接受输入。

另一个有趣的地方是 module_permission 函数。这个函数每当进程试图对/proc 文件进行处理时调用，它可以决定是否允许访问。目前这个函数只定义在操作和当前使用的 uid（当前可用的是一个指针指向一个当前运行进程的信息的结构）的基础上，但是它可以在我们希望的任何事物的基础上定义，比如其他进程正在对文件做的操作，日期时间或者接收到的最后一个输入。

使用 put_usr 和 get_user 的原因是 Linux 的内存是分段的（在 Intel 结构下，其他系列的处理器下可能不同）。这意味着一个指针本身不代表内存中的一个唯一地址，而是段中的一个地址，所以你还需要知道哪一个段可以使用它。内核占有一个段，每个进程都各占有一个段。

一个进程可以访问的唯一的段就是它自己拥有的那个，所以当你写作为进程运行的程序时可以不用关心段的问题。如果你要写内核模块，一般你希望访问内核的段，这由系统自动处理。然而，如果内存缓冲区的内容需要在当前运行的进程和内核之间传递时，内核函数会接到在此进程段里的指向内存缓冲区的一个指针。Put_user 和 get_user 允许你访问那块内存。

ex procfs.c

```
/* procfs.c -    create a "file" in /proc, which allows
  * both input and output. */

/* Copyright (C) 1998-1999 by Ori Pomerantz */


/* The necessary header files */

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

```
/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Necessary because we use proc fs */
#include <linux/proc_fs.h>


/* In 2.2.3 /usr/include/linux/version.h includes a
  * macro for this, but 2.0.35 doesn't - so I add it
  * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>    /* for get_user and put_user */
#endif

/* 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];


/* 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 */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t module_output(
        struct file *file,      /* The file read */
        char *buf, /* The buffer to put data to (in the
                              * user segment) */
        size_t len,    /* The length of the buffer */
        loff_t *offset) /* Offset in the file - ignore */
#else
```

```
static int module_output(
        struct inode *inode, /* The inode read */
        struct file *file,      /* The file read */
        char *buf, /* The buffer to put data to (in the
                              * user segment) */
        int len)    /* The length of the buffer */
#endif
{
   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<len && message[i]; i++)
      put_user(message[i], buf+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" */
}


/* This function receives input from the user when the
  * user writes to the /proc file. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
```

```
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 */
#else
static int module_input(
        struct inode *inode, /* The file's inode */
        struct file *file,      /* The file itself */
        const char *buf,        /* The buffer with the input */
        int length)             /* The buffer's length */
#endif
{
   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++)
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
        get_user(Message[i], buf+i);
   /* In version 2.2 the semantics of get_user changed,
     * it not longer returns a character, but expects a
     * variable to fill up as its first argument and a
     * user segment pointer to fill it from as the its
     * second.
     *
     * The reason for this change is that the version 2.2
     * get_user can also read an short or an int. The way
     * it knows the type of the variable it should read
     * is by using sizeof, and for that it needs the
     * variable itself.
     */
#else
        Message[i] = get_user(buf+i);
#endif
   Message[i] = '\0';    /* we want a standard, zero
                                * terminated string */

   /* We need to return the number of input characters
     * used */
   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)
{
    /* 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)
{
    MOD_INC_USE_COUNT;

    return 0;
}


/* The file is closed - again, interesting only because
  * of the reference count. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
int module_close(struct inode *inode, struct file *file)
#else
```

```
void module_close(struct inode *inode, struct file *file)
#endif
{
   MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
   return 0;    /* success */
#endif
}


/* 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 =
   {
      NULL,    /* lseek */
      module_output,    /* "read" from the file */
      module_input,      /* "write" to the file */
      NULL,    /* readdir */
      NULL,    /* select */
      NULL,    /* ioctl */
      NULL,    /* mmap */
      module_open,       /* Somebody opened the file */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      NULL,      /* flush, added here in version 2.2 */
#endif
      module_close,       /* Somebody closed the file */
      /* etc. etc. etc. (they are all given in
        * /usr/include/linux/fs.h). Since we don't put
        * anything here, the system will keep the default
        * data, which in Unix is zeros (NULLs when taken as
        * pointers). */
   };



/* 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 =
    {
        &File_Ops_4_Our_Proc_File,
        NULL, /* create */
        NULL, /* lookup */
        NULL, /* link */
        NULL, /* unlink */
        NULL, /* symlink */
        NULL, /* mkdir */
        NULL, /* rmdir */
        NULL, /* mknod */
        NULL, /* rename */
        NULL, /* readlink */
        NULL, /* follow_link */
        NULL, /* readpage */
        NULL, /* writepage */
        NULL, /* bmap */
        NULL, /* truncate */
        module_permission /* check for permissions */
    };


/* Directory entry */
static struct proc_dir_entry Our_Proc_File =
    {
        0, /* Inode number - ignore, it will be filled by
              * proc_register[_dynamic] */
        7, /* Length of the file name */
        "rw_test", /* The file name */
        S_IFREG | S_IRUGO | S_IWUSR,
        /* File mode - this is a regular file which
          * can be read by its owner, its group, and everybody
          * else. Also, its owner can write to it.
          *
          * Actually, this field is just for reference, it's
          * module_permission that does the actual check. It
          * could use this field, but in our implementation it
          * doesn't, for simplicity. */
```

```
        1,    /* Number of links (directories where the
                * file is referenced) */
        0, 0,    /* The uid and gid for the file -
                    * we give it to root */
        80, /* The size of the file reported by ls. */
        &Inode_Ops_4_Our_Proc_File,
        /* A pointer to the inode structure for
          * the file, if we need it. In our case we
          * do, because we need a write function. */
        NULL
        /* The read function for the file. Irrelevant,
          * because we put it in the inode structure above */
    };



/* Module initialization and cleanup ******************** */

/* Initialize the module - register the proc file */
int init_module()
{
    /* Success if proc_register[_dynamic] is a success,
      * failure otherwise */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    /* In version 2.2, proc_register assign a dynamic
      * inode number automatically if it is zero in the
      * structure , so there's no more need for
      * proc_register_dynamic
      */
    return proc_register(&proc_root, &Our_Proc_File);
#else
    return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif
}


/* Cleanup - unregister our file from /proc */
void cleanup_module()
{
    proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

5．和设备文件对话（写和 IOCTLS）

设备文件是用来代表物理设备的。多数物理设备是用来进行输出或输入的，所以必须由某种机制使得内核中的设备驱动从进程中得到输出送给设备。这可以通过打开输出设备文件并且写入做到，就想写入一个普通文件。在下面的例子里，这由 device_write 实现。

这不是总能奏效的。设想你与一个连向 modem 的串口（技是你有一个内猫，从 CPU 看来它也是作为一个串口实现，所以你不需要认为这个设想太困难）。最自然要做的事情就是使用设备文件把内容写到 modem 上（无论用 modem 命令还是电话线）或者从 modem 读信息（同样可以从 modem 命令回答或者通过电话线）。但是这留下的问题是当你需要和串口本身对话的时候需要怎样做？比如发送数据发送和接收的速率。

回答是 Unix 使用一个叫做 ioctl(input output control 的简写)的特殊函数。每个设备都有自己的 ioctl 命令，这个命令可以是 ioctl 读的，也可以是写的，也可以是两者都是或都不是。Ioctl 函数由三个参数调用：适当设备的描述子，ioctl 数，和一个长整型参数，可以赋予一个角色用来传递任何东西。

Ioctl 数对设备主码、ioctl 类型、编码、和参数的类型进行编码。Ioctl 数通常在头文件由一个宏调用（_IO，_IOR，_IOW 或_IOWR——决定于类型）。这个头文件必须包含在使用 ioctl（所以它们可以产生正确的 ioctl's）程序和内核模块（所以它可以理解）中。在下面的例子里，这个头文件是 chardev.h，使用它的程序是 ioctl.c。

如果你希望在你自己的内核模块中使用 ioctl's，最好去接受一分正式的 ioctl 职位，这样你就可以得到别人的 ioctl's，或者他们得到你，你就可以知道哪里出了错误。如果想得到更多的信息，到'documentation/ioctl-number.txt'中查看内核源文件树。

ex chardev.c

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


/* Copyright (C) 1998-99 by Ori Pomerantz */



/* The necessary header files */

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

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif
```

```
/* For character devices */

/* The character device definitions are here */
#include <linux/fs.h>

/* A wrapper which does next to nothing at
  * at present, but may help for compatibility
  * with future versions of Linux */
#include <linux/wrapper.h>


/* Our own ioctl numbers */
#include "chardev.h"


/* In 2.2.3 /usr/include/linux/version.h includes a
  * macro for this, but 2.0.35 doesn't - so I add it
  * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>    /* for get_user and put_user */
#endif



#define SUCCESS 0


/* Device Declarations ********************************* */


/* The name for our device, as it will appear in
  * /proc/devices */
#define DEVICE_NAME "char_dev"


/* The maximum length of the message for the device */
#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 function 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;

   /* If this was a process, we would have had to be
    * more careful here, because one process might have
    * checked Device_Open right before the other one
    * tried to increment it. However, we're in the
    * kernel, so we're protected against context switches.
    *
    * This is NOT the right attitude to take, because we
    * might be running on an SMP box, but we'll deal with
    * SMP in a later chapter.
    */

   Device_Open++;

   /* Initialize the message */
   Message_Ptr = Message;
```

```
   MOD_INC_USE_COUNT;

   return SUCCESS;
}


/* This function is called when a process closes the
  * device file. It doesn't have a return value because
  * it cannot fail. Regardless of what else happens, you
  * should always be able to close a device (in 2.0, a 2.2
  * device file could be impossible to close). */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static int device_release(struct inode *inode,
                                        struct file *file)
#else
static void device_release(struct inode *inode,
                                         struct file *file)
#endif
{
#ifdef DEBUG
   printk ("device_release(%p,%p)\n", inode, file);
#endif

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

   MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
   return 0;
#endif
}



/* This function is called whenever a process which
  * has already opened the device file attempts to
  * read from it. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_read(
        struct file *file,
        char *buffer, /* The buffer to fill with the data */
        size_t length,          /* The length of the buffer */
        loff_t *offset) /* offset to the file */
```

```
#else
static int device_read(
        struct inode *inode,
        struct file *file,
        char *buffer,      /* The buffer to fill with the data */
        int length)        /* The length of the buffer
                                * (mustn't write beyond that!) */
#endif
{
   /* 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. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_write(struct file *file,
                                        const char *buffer,
                                        size_t length,
                                        loff_t *offset)
#else
static int device_write(struct inode *inode,
                                    struct file *file,
                                    const char *buffer,
                                    int length)
#endif
{
   int i;

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

   for(i=0; i<length && i<BUF_LEN; i++)
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      get_user(Message[i], buffer+i);
#else
      Message[i] = get_user(buffer+i);
#endif

   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,
        struct file *file,
        unsigned int ioctl_num,/* The number of the ioctl */
        unsigned long ioctl_param) /* The parameter to it */
{
   int i;
   char *temp;
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
   char ch;
#endif

   /* 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 */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
         get_user(ch, temp);
         for (i=0; ch && i<BUF_LEN; i++, temp++)
            get_user(ch, temp);
#else
         for (i=0; get_user(temp) && i<BUF_LEN; i++, temp++)
       ;
#endif

         /* Don't reinvent the wheel - call device_write */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
         device_write(file, (char *) ioctl_param, i, 0);
#else
         device_write(inode, file, (char *) ioctl_param, i);
#endif
         break;

      case IOCTL_GET_MSG:
```

```
            /* Give the current message to the calling
             * process - the parameter we got is a pointer,
             * fill it. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
            i = device_read(file, (char *) ioctl_param, 99, 0);
#else
            i = device_read(inode, file, (char *) ioctl_param,
                                    99);
#endif
            /* Warning - we assume here the buffer length is
             * 100. If it's less than that we might overflow
             * the buffer, causing the process to core dump.
             *
             * The reason we only allow up to 99 characters is
             * that the NULL which terminates the string also
             * needs room. */

            /* 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 = {
    NULL,      /* seek */
    device_read,
```

```
    device_write,
    NULL,      /* readdir */
    NULL,      /* select */
    device_ioctl,      /* ioctl */
    NULL,      /* mmap */
    device_open,
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    NULL,    /* flush */
#endif
    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 = module_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 = module_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);
}
ex 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
```

ex 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.
 */

/* Copyright (C) 1998 by Ori Pomerantz */
```

```
/* 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);
}
```

6．启动参数

在以前的许多例子里，我们要把一些东西强制地写入内核模块，比如/proc 文件名或设备主码，以至我们可以用 ioctl’s 处理它。这样句违背了 Unix 以及 Linux 的原则：写用户可以自由设定的灵活程序。

在程序或者内核模块启动之前通知它一些消息是通过命令行参数做到的。在内核模块的情况下，我们没有 argc 和 argv 参数，而是有更好的东西。我们可以在内核模块里定义全局变量，insmod 会给我们赋值。

在这个内核模块中，我们定义了两个变量：str1 和 str2。你需要做的只是编译内核模块，然后运行 str1=xxx str2=yyy。当调用 init_module 时，str1 将指向串 xxx，str2 将指向串 yyy。

在 2.0 版对这些参数没有类型检查。如果 str1 和 str2 的第一个字符是数字，内核就会把这些变量赋为整数，而不是指向串的指针。这在实际情况中你一定要检查类型。

另一方面，在 2.2 版本中，你可以使用宏 MACRO_PARM 告诉 insmod 你需要一个参数，它的名字和类型。这样解决了类型问题，并且允许内核模块接收以数字开始的串。

ex **param.c**

```
/* param.c
 *
 * Receive command line parameters at module installation
 */

/* Copyright (C) 1998-99 by Ori Pomerantz */




/* The necessary header files */

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

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif


#include <stdio.h>    /* I need NULL */


/* In 2.2.3 /usr/include/linux/version.h includes a
```

```
  * macro for this, but 2.0.35 doesn't - so I add it
  * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



/* Emmanuel Papirakis:
  *
  * Prameter names are now (2.2) handled in a macro.
  * The kernel doesn't resolve the symbol names
  * like it seems to have once did.
  *
  * To pass parameters to a module, you have to use a macro
  * defined in include/linux/modules.h (line 176).
  * The macro takes two parameters. The parameter's name and
  * it's type. The type is a letter in double quotes.
  * For example, "i" should be an integer and "s" should
  * be a string.
  */


char *str1, *str2;


#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
MODULE_PARM(str1, "s");
MODULE_PARM(str2, "s");
#endif


/* Initialize the module - show the parameters */
int init_module()
{
   if (str1 == NULL || str2 == NULL) {
      printk("Next time, do insmod param str1=<something>");
      printk("str2=<something>\n");
   } else
      printk("Strings:%s and %s\n", str1, str2);

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
   printk("If you try to insmod this module twice,");
   printk("(without rmmod'ing\n");
```

```
    printk("it first), you might get the wrong");
    printk("error message:\n");
    printk("'symbol for parameters str1 not found'.\n");
#endif

    return 0;
}


/* Cleanup */
void cleanup_module()
{
}
```

7．系统调用

到此为止，我们做的事情就是使用定义好的内核机制来注册/proc 文件和设备句柄。这在做内核常规处理的事情时是很理想的。但是如果你希望做一些非常规的事情、改变系统的行为的时候该怎么办呢？这就必须依靠自己。

这就是内核编程变得危险的地方。在写下面的例子的时候，我关闭了 open 系统调用。这意味着我不能打开任何文件，不能运行任何程序，而且不能关闭计算机。我必须拉住电源开关。幸运的是，没有文件丢失。为确保你也不会丢失任何文件，在做 insmod 以及 rmmod 前请执行 sync 权限，

忘记/proc 文件，忘记设备文件。它们只是不重要的细节。真正的同内核通信的过程机制是被所有进程公用的，这就是系统调用。当一个进程请求内核服务时（比如打开文件、创建一个新进程或者要求更多内存），就需要使用这个机制。如果你想用比较有趣的方法改变内核行为，这就是你所需要的。另外，如果你希望看到程序使用了哪一个系统调用，运行 strace <command> <arguments>。

一般的，进程是不能访问内核的。它不能访问内核所占内存空间也不能调用内核函数。CPU 硬件决定了这些（这就是为什么它被称作“保护模式”）。系统调用是这些规则的一个例外。其原理是进程先用适当的值填充寄存器，然后调用一个特殊的指令，这个指令会跳到一个事先定义的内核中的一个位置（当然，这个位置是用户进程可读但是不可写的）。在 Intel CPU 中，这个由中断 0x80 实现。硬件知道一旦你跳到这个位置，你就不是在限制模式下运行的用户，而是作为操作系统的内核——所以你就可以为所欲为。

进程可以跳转到的内核位置叫做 sysem_call。这个过程检查系统调用号，这个号码告诉内核进程请求哪种服务。然后，它查看系统调用表(sys_call_table)找到所调用的内核函数入口地址。接着，就调用函数，等返回后，做一些系统检查，最后返回到进程（或到其他进程，如果这个进程时间用尽）。如果你希望读这段代码，它在源文件目录/<architecture>/kernel/entry.S，Entry(system_call)的下一行。

所以，如果我们希望改变某个系统调用的工作方式，我们需要写我们自己的函数（通常是加一点我们自己的代码然后调用原来的函数）来实现，然后改变 sys_call_table 中的指针使其指向我们的函数。因为我们可能以后会删除，而且不希望系统处在不稳定状态，所以在 cleanup_module 中保存该表的原来状态很重要。

这里的源代码是一个这样的核心模块的例子。我们希望“窥探”一个用户，每当这个用户打开一个文件是就 printk 一条消息。为达到这个目的，我们把打开文件的系统调用替换为我们自己的函数，our_sys_open。这个函数检查当前进程的 uid（用户的 id），如果它等于我们要窥探的 uid，就调用 printk 来显示所打开文件的文件名。然后，可以用任何一种方法，用同样的参数调用原来的 open 函数，或者真正打开文件。

Init_module 函数把 sys_call_table 中的适当地址上的内容替换，把原来的指针保存在一个变量里。Cleanup_module 函数用这些变量恢复所有的东西。这种方法是危险的，因为两个内核模块可能改变了同一个系统调用。设想我们由两个内核模块，A 和 B。A 的 open 系统调用是 A_open，B 的 open 系统调用是 B_open。现在，如果 A 插入内核，系统调用将被替换为 A_open，当完成以后调用 sys_open。然后，B 被插入内核，把系统调用替换为 B_open，而完成的时候，它将会调用它认为原始的系统调用的 A_open，。

那么，如果 B 被首先删除，不会出现任何错误——它只是把系统调用恢复成 A_open，A_open 再去调用原始的的系统调用。然而，如果先删除 A，再删除 B，系统就会崩溃。A 的删除将会把系统调用恢复成 sys_open，而把 B 切换出了循环。然后，当 B 被删除时，将会把系统调用恢复成 A_open，但是 A_open 已经不在内存。初看来，似乎我们可以通过检

查系统调用是否等于我们的 open 函数来解决这个问题，如果是就不要改变它（这样 B 被删除的时候就不会改变系统调用），但是这样会引起一个更加恶劣的问题。当 A 被删除时，它看到系统调用被改成了 B_open 而不再指向 A_open，所以在它被删除时就不会恢复 sys_open。不幸的是，B_open 仍然试图恢复 A_open，但它已不再内存，这样，即使没有删除 B 系统也会崩溃。

我可以提出两个方法来解决这个问题。第一个方法是把调用恢复成原始值，sys_open。不幸的是 sys_open 不是在/proc/ksyms 中的内核系统表中的一部分，所以我们不能访问它。另一个解决办法是使用索引计数器来阻止 root 去 rmmod 这个模块，一旦它被装载。这在生产性模块中是好的，但是对教学里中不是很好——这就是为什么我不在这里这样做。

ex **syscall.c**

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


/* Copyright (C) 1998-99 by Ori Pomerantz */


/* The necessary header files */

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

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <sys/syscall.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>




/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
```

```
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>
#endif



/* 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
  */
extern void *sys_call_table[];



/* UID we want to spy on - will be filled from the
  * command line */
int uid;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
MODULE_PARM(uid, "i");
#endif

/* 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);



/* For some reason, in 2.2.3 current->uid gave me
  * zero, not the real user ID. I tried to find what went
```

```
  * wrong, but I couldn't do it in a short time, and
  * I'm lazy - so I'll just use the system call to get the
  * uid, the way a process would.
  *
  * For some reason, after I recompiled the kernel this
  * problem went away.
  */
asmlinkage int (*getuid_call)();



/* 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 == getuid_call()) {
     /* getuid_call is the getuid system call,
       * which gives the uid of the user who
       * ran the process which called the system
       * call we got */

       /* Report the file, if relevant */
       printk("Opened file by %d: ", uid);
       do {
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
          get_user(ch, filename+i);
#else
```

```
            ch = get_user(filename+i);
#endif
            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);

    /* Get the system call for getuid */
    getuid_call = sys_call_table[__NR_getuid];

    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;
}
```

8．阻塞进程

如果有人让你做你一时做不到的事情你会怎么办呢？如果你是个人被另一个人打扰，你唯一可以做的就是对他说：“现在不行，我很忙，走开！”但是如果你是内核模块，被进程打扰，你就有另一种选择。你可以让这个进程去挂起直到你可以为之提供服务。毕竟，进程是在不停的被内核挂起或唤醒（这就是多个进程看上去同时在一个处理器上运行的方法）。

这个内核模块就是一个这样的例子。这个文件（称作/proc/sleep）在一个时刻只能被一个进程打开。如果这个文件已经被打开，内核模块就调用 module_interruptible_sleep_on（注 8.1）。这个函数把任务（一个任务是一个内核数据结构，它包含进程以及它所在系统调用的信息）的状态改变成 TASK_INTERRUPTIBLE，表示直到被唤醒任务不会运行，并且把它加入到 WaitQ——等待访问文件的任务队列。那么，这个函数调用调度器进行上下文切换到其他要使用 CPU 的进程。

当进程完成对文件的处理后，关闭该文件并且调用 module_ close。这个函数唤醒所有队列中的进程（还没有一个机制唤醒其中一个）。然后返回，刚才关闭文件的进程可以继续运行。调度器及时决定哪个进程已经完成，并且把 CPU 的控制给另一个进程。同时，队列中的某个进程将会从调度器那里得到对 CPU 的控制。它正在对 module_interruptible_sleep_on 的调用后开始。然后它可以设置一个全局变量告诉别的进程这个文件还打开着，正在继续它的生命。当别的进程有得到 CPU 的机会时，它们会看到这个全局变量，然后就重新挂起。

为使生命更加精彩，module_close 并没有对唤醒等待访问文件的进程进行垄断。一个象 Ctrl-C（SIGINT）之类的信号同样可以唤醒进程。在这种情况下，我们希望立即返回-EINTR。这是很重要的，比如用户可以在进程接到文件前杀死进程。

还有一点需要记住。有些时候进程不希望被挂起，它们希望立刻得到它们要的东西，或者被告知不能做到。这样的进程在打开文件时使用 O_NONBLOCK 标志。内核在遇到其他方面的挂起进程的操作（比如本例中的打开文件）时要返回一个错误码-ERROR 作为回应。程序 cat_noblock 就可以用来使用标志 O_NONBLOCK 打开文件，它可以在本章源程序目录中找到。

ex **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 */


/* Copyright (C) 1998-99 by Ori Pomerantz */


/* The necessary header files */

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

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
```

```
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Necessary because we use proc fs */
#include <linux/proc_fs.h>

/* For putting processes to sleep and waking them up */
#include <linux/sched.h>
#include <linux/wrapper.h>



/* In 2.2.3 /usr/include/linux/version.h includes a
  * macro for this, but 2.0.35 doesn't - so I add it
  * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif


#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>    /* for get_user and put_user */
#endif


/* 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];


/* 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 */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t module_output(
        struct file *file,      /* The file read */
        char *buf, /* The buffer to put data to (in the
                              * user segment) */
        size_t len,    /* The length of the buffer */
```

```
        loff_t *offset) /* Offset in the file - ignore */
#else
static int module_output(
        struct inode *inode, /* The inode read */
        struct file *file,      /* The file read */
        char *buf, /* The buffer to put data to (in the
                              * user segment) */
        int len)    /* The length of the buffer */
#endif
{
   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. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
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 */
#else
static int module_input(
        struct inode *inode, /* The file's inode */
        struct file *file,      /* The file itself */
```

```
        const char *buf,          /* The buffer with the input */
        int length)               /* The buffer's length */
#endif
{
   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++)
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
         get_user(Message[i], buf+i);
#else
   Message[i] = get_user(buf+i);
#endif
/* 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 */
static struct wait_queue *WaitQ = NULL;


/* 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 MOD_INC_USE_COUNT
     * because if a process is in the loop, which is
```

```
     * within the kernel module, the kernel module must
     * not be removed. */
    MOD_INC_USE_COUNT;

    /* If the file is already open, wait until it isn't */
    while (Already_Open)
    {
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
        int i, is_sig=0;
#endif

        /* 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 */
        module_interruptible_sleep_on(&WaitQ);

        /* 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.
  */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)

        for(i=0; i<_NSIG_WORDS && !is_sig; i++)
```

```
            is_sig = current->signal.sig[i] &
              ~current->blocked.sig[i];
        if (is_sig) {
#else
        if (current->signal & ~current->blocked) {
#endif
            /* It's important to put MOD_DEC_USE_COUNT 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. */
            MOD_DEC_USE_COUNT;
            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 */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
int module_close(struct inode *inode, struct file *file)
#else
void module_close(struct inode *inode, struct file *file)
#endif
{
    /* 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. */
```

```
    module_wake_up(&WaitQ);

    MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    return 0;    /* success */
#endif
}



/* 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)
{
    /* 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 =
   {
      NULL,    /* lseek */
      module_output,    /* "read" from the file */
      module_input,     /* "write" to the file */
      NULL,    /* readdir */
      NULL,    /* select */
      NULL,    /* ioctl */
      NULL,    /* mmap */
      module_open,/* called when the /proc file is opened */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      NULL,     /* flush */
#endif
      module_close           /* called when it's classed */
   };



/* 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 =
   {
      &File_Ops_4_Our_Proc_File,
      NULL, /* create */
      NULL, /* lookup */
      NULL, /* link */
      NULL, /* unlink */
      NULL, /* symlink */
      NULL, /* mkdir */
      NULL, /* rmdir */
      NULL, /* mknod */
      NULL, /* rename */
      NULL, /* readlink */
      NULL, /* follow_link */
      NULL, /* readpage */
      NULL, /* writepage */
      NULL, /* bmap */
      NULL, /* truncate */
      module_permission /* check for permissions */
```

```
    };

/* Directory entry */
static struct proc_dir_entry Our_Proc_File =
    {
        0, /* Inode number - ignore, it will be filled by
              * proc_register[_dynamic] */
        5, /* Length of the file name */
        "sleep", /* The file name */
        S_IFREG | S_IRUGO | S_IWUSR,
        /* File mode - this is a regular file which
          * can be read by its owner, its group, and everybody
          * else. Also, its owner can write to it.
          *
          * Actually, this field is just for reference, it's
          * module_permission that does the actual check. It
          * could use this field, but in our implementation it
          * doesn't, for simplicity. */
        1,    /* Number of links (directories where the
                 * file is referenced) */
        0, 0,    /* The uid and gid for the file - we give
                    * it to root */
        80, /* The size of the file reported by ls. */
        &Inode_Ops_4_Our_Proc_File,
        /* A pointer to the inode structure for
          * the file, if we need it. In our case we
          * do, because we need a write function. */
        NULL    /* The read function for the file.
                   * Irrelevant, because we put it
                   * in the inode structure above */
    };



/* Module initialization and cleanup ********** */


/* Initialize the module - register the proc file */
int init_module()
{
    /* Success if proc_register_dynamic is a success,
      * failure otherwise */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    return proc_register(&proc_root, &Our_Proc_File);
```

```
#else
   return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif

   /* proc_root is the root directory for the proc
    * fs (/proc). This is where we want our file to be
    * located.
    */
}


/* 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()
{
   proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

9．替换 printk’s

在开始（第 1 章）的时候，我们说过 X 与内核模块编程并不混合。这开发内核的时候这是正确的，但是在实际应用中我们希望把消息送到给模块的命令发来的任何一个 tty（注 9.1）。这在内核模块被释放时确认错误是很重要的，因为它将会在所有内核中使用。

这样做的方法是使用当前的概念，一个指向当前运行任务的指针，从而得到当前任务的 tty 结构。然后，我们到 tty 结构里寻找一个指向写串函数的指针，我们用这个函数把一个串写进 tty。

ex printk.c

```
/* printk.c - send textual output to the tty you're
 * running on, regardless of whether it's passed
 * through X11, telnet, etc. */


/* Copyright (C) 1998 by Ori Pomerantz */


/* The necessary header files */

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

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Necessary here */
#include <linux/sched.h>       /* For current */
#include <linux/tty.h>         /* For the tty declarations */


/* Print the string to the appropriate tty, the one
 * the current task uses */
void print_string(char *str)
{
   struct tty_struct *my_tty;

   /* The tty for the current task */
   my_tty = current->tty;
```

```
/* If my_tty is NULL, it means that the current task
  * has no tty you can print to (this is possible, for
  * example, if it's a daemon). In this case, 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 memory segment
      * or the kernel's memory segment.
      *
      * The function's first parameter is the tty to
      * write to, because the    same function would
      * normally be used for all tty's of a certain type.
      * The second parameter controls whether the
      * function receives a string from kernel memory
      * (false, 0) or from user memory (true, non zero).
      * The third parameter is a pointer to a string,
      * and the fourth parameter is the length of
      * the string.
      */
    (*(my_tty->driver).write)(
            my_tty, /* The tty itself */
            0, /* We don't take the string from user space */
    str, /* String */
    strlen(str));    /* Length */

    /* ttys were originally hardware devices, which
      * (usually) adhered strictly to the ASCII standard.
      * According to ASCII, to move to a new line you
      * need two characters, a carriage return and a
      * line feed. In Unix, on the other hand, 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.
      *
      * BTW, this is the reason why the text file
      *    is different between Unix and Windows.
      * In CP/M and its derivatives, such as MS-DOS and
      * Windows, the ASCII standard was strictly
      * adhered to, and therefore a new line requires
```

```
         * both a line feed and a carriage return.
         */
        (*(my_tty->driver).write)(
            my_tty,
            0,
            "\015\012",
            2);
    }
}


/* Module initialization and cleanup ****************** */


/* Initialize the module - register the proc file */
int init_module()
{
    print_string("Module Inserted");

    return 0;
}


/* Cleanup - unregister our file from /proc */
void cleanup_module()
{
    print_string("Module Removed");
}
```

10．调度任务

经常地，我们有必须定时做或者经常做的“家务事”。如果这个任务由一个进程完成，我们可以把通过把它放入 crontab 文件而做到。如果这个任务由一个内核模块完成，我们有两种可能的选择。第一种是把一个进程放入 crontab 文件，它将在必要的时候通过一个系统调用唤醒模块，比如打开一个文件。然而，这样做时非常低效的，我们需要运行一个 crontab 外的新进程，把一个新的执行表读入内存，而所有这些只是为了唤醒一个内存中的内核模块。

我们不需要这样做。我们可以创建一个函数，在每个时间中断时被调用。方法是创建一个任务，包含在一个结构体 tq_struct 里，在此结构中包含一个指向函数入口地址的指针。然后，我们使用 queue_task 把这个任务放入一个叫做 tq_timer 的任务列表中，这是一个在下次时间中断时要执行的任务列表。因为我们希望这个函数被持续执行，我们需要在每次调用厚把它放回 tq_timer 中以备下次时间中断。

这里还有一点需要记住。当一个模块被 rmmod 删除时，首先他的索引计数器被检查。如果是 0，就调用 module_cleanup。然后，这个模块以及它的所有函数都从内存中删除。没有人检查是否时钟的任务列表仍然包含指向这些函数的指针，而现在已不可用。很久以后（从计算机看来，在人的眼睛里是很短的，可能是百分之一秒），内核有了一个时钟中断，试图调用任务列表中的所有函数。不幸的是，这个函数已不存在。在多数情况下，它所在的内存还未被使用，而你得到了一个极端错误的信息。但是，如果有别的代码出在相同的地址，情况会非常糟糕。不幸的是，我们没有一个从任务列表中注销一个任务的方法。

既然 cleanup_module 函数不能返回一个错误马（它是 void 型函数），那么解决方法是就不要让它返回。而是调用 sleep_on 或 module_sleep_on（注 10.1）把 rmmod 进程挂起。在此之前，它设置一个变量通知在时钟中断时调用的函数停止附加自己。那么，在下次时钟中断时，rmmod 进程将被唤醒，而我们的函数已经不在队列中，这样就可以很安全的删除模块。

ex **sched.c**

```
/* sched.c - scheduale a function to be called on
  * every timer interrupt. */



/* Copyright (C) 1998 by Ori Pomerantz */



/* The necessary header files */

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

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif
```

```
/* Necessary because we use the proc fs */
#include <linux/proc_fs.h>

/* We scheduale tasks here */
#include <linux/tqueue.h>

/* We also need the ability to put ourselves to sleep
  * and wake up later */
#include <linux/sched.h>

/* In 2.2.3 /usr/include/linux/version.h includes a
  * macro for this, but 2.0.35 doesn't - so I add it
  * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



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


/* This is used by cleanup, to prevent the module from
  * being unloaded while intrpt_routine is still in
  * the task queue */
static struct wait_queue *WaitQ = NULL;

static void intrpt_routine(void *);


/* The task queue structure for this task, from tqueue.h */
static struct tq_struct Task = {
   NULL,      /* Next item in list - queue_task will do
                 * this for us */
   0,           /* A flag meaning we haven't been inserted
                 * into a task queue yet */
   intrpt_routine, /* The function to run */
   NULL        /* The void* parameter for that function */
};
```

```
/* 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 (WaitQ != NULL)
      wake_up(&WaitQ);      /* Now cleanup_module can return */
   else
      /* Put ourselves back in the task queue */
      queue_task(&Task, &tq_timer);
}




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

   /* This is 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 was 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;
}


struct proc_dir_entry Our_Proc_File =
    {
        0, /* Inode number - ignore, it will be filled by
            * proc_register_dynamic */
        5, /* Length of the file name */
        "sched", /* The file name */
        S_IFREG | S_IRUGO,
        /* File mode - this is a regular file which can
         * be read by its owner, its group, and everybody
         * else */
        1,    /* Number of links (directories where
               * the file is referenced) */
        0, 0,    /* The uid and gid for the file - we give
                  * it to root */
        80, /* The size of the file reported by ls. */
        NULL, /* functions which can be done on the
                 * inode (linking, removing, etc.) - we don't
                 * support any. */
        procfile_read,
        /* The read function for this file, the function called
         * when somebody tries to read something from it. */
        NULL
        /* We could have here a function to fill the
         * file's inode, to enable us to play with
         * permissions, ownership, etc. */
    };


/* Initialize the module - register the proc file */
int init_module()
{
    /* Put the task in the tq_timer task queue, so it
```

```
     * will be executed at next timer interrupt */
    queue_task(&Task, &tq_timer);

    /* Success if proc_register_dynamic is a success,
     * failure otherwise */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
    return proc_register(&proc_root, &Our_Proc_File);
#else
    return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif
}


/* Cleanup */
void cleanup_module()
{
    /* Unregister our /proc file */
    proc_unregister(&proc_root, Our_Proc_File.low_ino);

    /* 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 tq_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. */
  sleep_on(&WaitQ);
}
```

11．中断处理程序

除最后一章外，目前我们在内核中所做的事情就是响应一个进程的请求，可能通过处理一个特殊文件、发送一个 ioctl 或进行一个系统调用。但是内核的工作不只是响应进程请求，另一个也是很重要的工作是向连接到机器的硬件发布命令或消息。

CPU 和其他硬件的交互有两种。第一种是 CPU 给硬件发命令。另一种是硬件需要告诉 CPU 一些东西。第二种交互叫做中断，它很难实现，因为它需要处理硬件，而不是 CPU。硬件设备通常由一个非常小的 ram，而如果你不在这个 ram 中的信息可用时把它们读出，它们就会丢失。

在 Linux 下，硬件中断叫做 IRQ（Interrupt Requests 的缩写）。有两种 IRQ，短类型和长类型。短 IRQ 需要很短的时间，在此期间机器的其他部分被锁定，而且没有其他中断被处理。一个长 IRQ 需要较长的时间，在此期间可能发生其他中断（但不是发自同一个设备）。如果可能的话，最好把一个中段声明为长类型。

如果 CPU 接到一个中断，它就会停止一切工作（除非它正在处理一个更重要的中断，在这种情况下要等到更重要的中断处理结束后才会处理这个中断），把相关的参数存储到栈里，然后调用中断处理程序。这意味着在中断处理程序本身中有些事情是不允许的，因为这时系统处在一个未知状态。解决这个问题的方法是让中断处理程序做需要马上做的事，通常是从硬件读取信息或给硬件发送信息，然后把对新信息的处理调度到以后去做（这叫做半底），返回。内核确保尽快调用半底，而当调用时，任何允许在内核模块中做的事情就都可以做了。

实现的方法是在接到相关的 IRQ（在 Intel 平台上有 16 个 IRQ）时调用中断处理程序。这个函数接到 IRQ 号码、函数名、标志、一个/proc/interrupts 的名字和传给中断处理程序的一个参数。标志中可以包括 SA_SHIRQ 来表明你希望和其他处理程序共享此 IRQ（通常很多设备公用一个 IRQ），或者一个 SA_INTERRUPT 表明这是一个紧急中断。这个函数仅在此 IRQ 没有其他处理程序或需要共享所有处理程序时才会成功运行。

那么，有了中断处理程序，我们就可以和硬件通信，并且可以使用 queue_task_irq 和 tq_immediate 和 mark_bh(BH_IMMEDIATE)来调度半底。在 2.0 版本中不能使用标准 queue_task 的原因是中断可能发生在某个 queue_task 的中间。我们需要 mark_bh 是因为早期的 Linux 版本只有一个长度为 32 的半底队列。现在它们中的一个（BH_IMMEDIATE）用于那些没有得到分配给它们的半底入口的驱动程序的半底链表。

11.1 Intel 结构上的键盘

注意：本章下面的内容都是完全针对 Intel 结构。如果你不是在 Intel 平台上的话，程序就不能工作，也不没有必要试图编译。

在写本章例子中的程序的时候我遇到一个问题。一方面，作为一个有用的例子它需要在每个人的机器上运行，得到正确的结果。另一方面，内核已经包含了所有普通设备的驱动程序，这些驱动程序和我要写的代码不一定能够共存。我找到的方法是给键盘中断写点东西，并且首先使正常的键盘中断无效。因为在内核源文件（drivers/char/keyboard.c）里它是作为一个静态符号被定义的，没有办法恢复它。在 insmod 这段代码前，在另一个终端 sleep 120 上做一次，如果要评估文件系统的话，需要 reboot。

这段代码把它自己绑定到 IRQ1 上，在 Interl 结构下这是键盘控制的 IRQ。然后，当它接到一个键盘中断时，读出键盘的状态（这是 inb(0x64)的目的）和由键盘返回的扫描码。然后，只要内核认为可以时，它就运行 got_char 给出键的编码（扫描码的前 7 位）和是否被

按下的信息（如果第 8 位是 0 则表示按下，是 1 表示释放）。

ex intrpt.c

```
/* intrpt.c - An interrupt handler. */


/* Copyright (C) 1998 by Ori Pomerantz */



/* The necessary header files */

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

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/sched.h>
#include <linux/tqueue.h>

/* We want an interrupt */
#include <linux/interrupt.h>

#include <asm/io.h>


/* In 2.2.3 /usr/include/linux/version.h includes a
  * macro for this, but 2.0.35 doesn't - so I add it
  * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



/* Bottom Half - 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
 * scheduales the bottom half to run when the kernel
 * considers it safe. */
void 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 unsigned char scancode;
   static struct tq_struct task =
           {NULL, 0, got_char, &scancode};
   unsigned char status;

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

   /* Scheduale bottom half to run */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
   queue_task(&task, &tq_immediate);
#else
   queue_task_irq(&task, &tq_immediate);
#endif
   mark_bh(IMMEDIATE_BH);
}



/* Initialize the module - register the IRQ handler */
int init_module()
{
   /* 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. */
    return request_irq(
        1,    /* The number of the keyboard IRQ on PCs */
        irq_handler,    /* our handler */
        SA_SHIRQ,
        /* 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.
         */
        "test_keyboard_irq_handler", NULL);
}


/* 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);
}
```

12．对称多处理

提高硬件性能的最简单（最便宜）的方法是在主板上增加 CPU。这可以让不同 CPU 做不同工作（非对称多处理）或者让它们并行运行，做相同工作（对称多处理，也叫 SMP）。有效的进行飞对称多处理需要有对计算机任务执行专业知识，但在一般操作系统比如 Linux 中这是不可知的。另一方面，对称多处理相对容易实现。所谓相对，就是说不是真的很容易。在对称多处理环境里，所有 CPU 共享同一内存，那么，在一个 CPU 上运行的代码会影响被另一个 CPU 使用的内存。你就不能确保你在某一行设定的变量在下一行仍然是原来的值——另一个 CPU 可能在你没看到的时候改变了它。显然，不能这样编程。

在进程编程里这不是一个问题，因为在某一时刻只有一个进程在处理机上。另一方面，内核可以运行在不同 CPU 上的不同进程调用。

在 2.0.x 版中，这不是个问题，因为整个内核在一个大的连环锁中。这就是说如果一个 CPU 在内核中，而有另一个 CPU 希望进入，比如因为一个系统调用，那么它必须等前一个 CPU 工作完成。这使 SMP 很安全，但是也很低效。

在 2.2.x 版中，几个 CPU 可以同时在 CPU 中。这是模块编写者需要注意的问题。我已经让人给我一个 SMP 盒，希望本书的下一版可以有更多关于 SMP 的介绍。

常见的错误

在我告诉你们进入实践写内核模块前，还有几个地方需要提醒一下。如果发生了不好的事情而我没有警告，请把问题报告给我并且索取您为此书给我的报酬的全额赔偿。

1．使用标准库。你不能那样做。在内核模块中你只能使用内核函数，它们就是在./proc/ksyms 中的那些。
2．作废中断。也许你需要在短时间内这样做，这是可以的。但是如果你没有重新重新使它们可用，那么你的系统就会死机，而你需要关机。
3．冒险。也许我不需要警告你这一条，但是我认为还是需要，以防万一。

2.0 和 2.2 版本的区别

以我对整个内核的模块的了解还不能写出它们所有的区别的文档。在转换这些例子（或者实际的说，是使用了 Emmanuel Papirakis 的改变）的过程中，我遇到了下面的区别。我把它们列出来帮助内核模块程序员，特别是那些学习过本书前面各版熟悉我使用的技术的程序员，来转化到新的版本。

1．**Asm/success.h** 如果你需要 put_user 或 get_user 你需要包含它。
2．**Get_user** 在 2.2 版中，get_user 接到指向我们的内存的指针和指向用此信息设置内核中的变量的指针。原因是 get_user 可以一次读 2 或 4 个字节，如果要读的变量有 2 或 4 个字节长的话。
3．**File_operations** 这个结构体现在在 open 和 close 之间有一个 flush 函数。
4．**Close in file_open** 在 2.2 版中，close 返回一个整型，所以它允许失败。
5．**Read and write in file_operations** 这个函数头改变了。它们现在返回一个 ssize_t 而不是整数，它们的参数列表也改变了。索引节点不再是一个参数，而文件中的偏移地址成了一个参数。
6．**Proc_register_dynamic** 这个函数已不存在。代替它的是，调用正常的 proc_register，把 0 写入这个结构体的索引节点域中。
7．**Signals** 任务结构中的信号量不再是一个 32 位整数，而是一个_NSIG_WORDS 整数数组。
8．**Queue_task_irq** 即使你要调度一个从中断处理程序中发生的任务，你也要使用 queue_task 而不是 queue_task_irq。
9．**Module Parameters** 你不需要把模块参数声明为全局变量了。在 2.2 中你必须要 MODULE_PARM 来声明它们的类型。这是一个大的改进，因为它使得模块可以接受以数字开始的字符串参数而不会混淆。
10．**Symmetrical Multi-Processing** 内核不再在一个大的连环锁内工作了，这意味着内核模块必须注意到 SMP。

除此以外

我可以很容易的在本书中加入几章。我可以增加一章关于创建新文件系统的内容，或者增加新的协议栈。我也可以增加对我们从来没接触到的内核机制的解释，比如捆绑导入或磁盘交互。

但是我没有。我写书的目的是要对神秘的内核编程进行启蒙，讲解最基本的技术。对于对内核编程有浓厚兴趣的人，我可以向他们推荐一个资源网站：http:.//jungla.dit.upm.es/~jmseyas/linux/kernel/hackers-docs.html。而且，正如 Linus 所说，学习内核的最好方法就是自己读源代码。

如果你希望更多的短内核模块的代码，我推荐 Phrack 杂志。即使你对安全性不感兴趣，尽管作为一个程序员你应该考虑安全性，那里有许多很好的例子可以告诉你可以在内核里做什么，这些例子都很短小，容易看懂。

我希望我为你成为更好的程序员提供了帮助，或者至少在技术中显示出了乐趣。如果你写出了有用的内核模块，我希望你能在 GPL 下发表，那么我也可以使用。

其他

Goods and Services

I hope nobody minds the shameless promotions here. They are all things which are likely to be of use to beginning Linux Kernel Module programmers.

Getting this Book in Print

The Coriolis group is going to print this book sometimes in the summer of '99. If this is already summer, and you want this book in print, you can go easy on your printer and buy it in a nice, bound form.

GNU GENERAL PUBLIC LICENSE

Printed below is the GNU General Public License (the *GPL* or *copyleft*), under which this book is licensed.

Version 2, June 1991

Copyright ©1989, 1991 Free Software Foundation, Inc. 675 Mass Ave, Cambridge, MA 02139, USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

PREAMBLE

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software-to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this

license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0.

This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The `Program', below, refers to any such program or work, and a `work based on the Program' means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term `modification'.) Each licensee is addressed as `you'.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1.

You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2.

You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a.

You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b.

You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third

parties under the terms of this License.

c.

If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3.

You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a.

Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b.

Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c.

Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the

major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4.

You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5.

You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6.

Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7.

If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot

impose that choice.
This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.
8.
If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9.
The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.
Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and `any later version', you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
10.
If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY
11.
BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM `AS IS' WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12.
IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED

INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

APPENDIX: HOW TO APPLY THESE TERMS TO YOUR NEW PROGRAMS

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the `copyright' line and a pointer to where the full notice is found.

one line to give the program's name and a brief idea of what it does. Copyright ©19yy name of author

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

The hypothetical commands show w and show c should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than show w and show c; they could even be mouse-clicks or menu items-whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a `copyright disclaimer' for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program Gnomovision (which makes passes at compilers) written by James Hacker.

***signature of Ty Coon*, 1 April 1989**

Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

注

1.1 我没有以 root 身份编译是因为用 root 做的事情越少，系统就越安全。我是做计算机安全工作的，所以比较偏执。

3.1 这是在 2.0 版本中。在 2.2 中如果把索引节点设为 0 这是自动完成的。

4.1 这两个的区别是文件操作是处理文件本身，而索引节点操作时处理引用文件的方法，比如创建连接。

5.1 注意这里读和写角色再次换位，在 ioctl 中读是向内核发信息，而写是从内核读信息。

5.2 这是不确切的。比如你不能通过 ioctl 传递一个结构体，但是可以传递指向它的指针。

6.1 不可能有。因为在 C 语言中目标文件只有全局变量的地址，而不是类型。这就是为什么头文件是必要的。

8.1 最简单的把文件保持打开状态的方法是用 tail –f 打开。

8.2 这是说进程仍在内核——对于进程来说，它发出 open 系统调用，而这个系统调用还没有返回。多数情况下，在发出系统调用和返回之间的时间内，进程不知道别人在使用 CPU。

8.3 这是因为我们使用了 module_interruptible_sleep_on。我们可以使用 module_sleep_on，但是会激怒那些控制程序被打扰的用户。

9.1 Teletype，最初是用来和 Unix 系统通信的键盘-打印机的结合体，现在成了 Unix 程序的文本流的概括，无论是物理终端、X 显示的一个 xterm、网络连接还是其他。

10.1 它们其实是一样的。

11.1 这是 Linux 发源的 Intel 结构上的标准方法。

11.2 queue_task_irq 是用一个全局锁来保护不受此影响，在 2.2 中没有 queue_task_irq，而 queue_task 是被锁保护的。

12.1 例外是线连进程，可以同时在几个 CPU 上运行。

12.2 表示可以在 SMP 中可以安全使用。