Linux Kernel Module Programming Guide

1999 Ori Pomerantz

Version 1.1.0, 26 April 1999.

This book is about writing Linux Kernel Modules. It is, hopefully, useful for programmers who know C and want to learn how to write kernel modules. It is written as an 'How-To' instruction manual, with examples of all of the important techniques.

Although this book touches on many points of kernel design, it is not supposed to fulfill that need — there are other books on this subject, both in print and in the Linux documentation project.

Names of all products herein are used for identification purposes only and are trademarks and/or registered trademarks of their respective owners. I make no claim of ownership or corporate association with the products or companies that own them.

Ori Pomerantz
Apt. #1032
2355 N Hwy 360
Grand Prairie
TX 75050
USA
E-mail: mpg@simple-tech.com

Contents

Chapter 0

Introduction

So, you want to write a kernel module. You know C, you've written a number of normal programs to run as processes, and now you want to get to where the real action is, to where a single wild pointer can wipe out your file system and a core dump means a reboot.

Well, welcome to the club. I once had a wild pointer wipe an important directory under DOS (thankfully, now it stands for the **D**ead **O**perating **S**ystem), and I don't see why living under Linux should be any safer.

Warning: I wrote this and checked the program under versions 2.0.35 and 2.2.3 of the kernel running on a Pentium. For the most part, it should work on other CPUs and on other versions of the kernel, as long as they are 2.0.x or 2.2.x, but I can't promise anything. One exception is chapter 11, which should not work on any architecture except for x86.

0.1 Who Should Read This

This document is for people who want to write kernel modules. Although I will touch on how things are done in the kernel in several places, that is not my purpose. There are enough good sources which do a better job than I could have done.

This document is also for people who know how to write kernel modules, but have not yet adapted to version 2.2 of the kernel. If you are such a person, I suggest you look at appendix A to see all the differences I encountered while updating the examples. The list is nowhere near comprehensive, but I think it covers most of the basic functionality and will be enough to get you started.

The kernel is a great piece of programming, and I believe that programmers should read at least some kernel source files and understand them. Having said that, I also believe in the value of playing with the system first and asking questions later. When I learn a new programming language, I don't start with reading the library code, but by writing a small 'hello, world' program. I don't see why playing with the kernel should be any different.

0.2 Note on the Style

I like to put as many jokes as possible into my documentation. I'm writing this because I enjoy it, and I assume most of you are reading this for the same reason. If you just want to get to the point, ignore all the normal text and read the source code. I promise to put all the important details in remarks.

0.3 Changes

0.3.1 New in version 1.0.1

1. **Changes section**, 0.3.
2. **How to find the minor device number**, 2.
3. **Fixed the explanation of the difference between character and device files**, 2
4. **Makefiles for Kernel Modules**, 1.1.
5. **Symmetrical Multiprocessing**, 12.
6. **A 'Bad Ideas' Chapter**, 13.

0.3.2 New in version 1.1.0

1. **Support for version 2.2 of the kernel**, all over the place.
2. **Multi kernel version source files**, 2.1.
3. **Changes between 2.0 and 2.2**, A.
4. **Kernel Modules in Multiple Source Files**, 1.2.
5. **Suggestion not to let modules which mess with system calls be rmmod'ed**, 7.

0.4 Acknowledgements

I'd like to thank Yoav Weiss for many helpful ideas and discussions, as well as for finding mistakes within this document before its publication. Of course, any remaining mistakes are purely my fault.

The TeX skeleton for this book was shamelessly stolen from the 'Linux Installation and Getting Started' guide, where the TeX work was done by Matt Welsh.

My gratitude to Linus Torvalds, Richard Stallman and all the other people who made it possible for me to run a high quality operating system on my computer and get the source code goes without saying (yeah, right — then why did I say it?).

0.4.1 For version 1.0.1

I couldn't list everybody who e-mailed me here, and if I've left you out I apologize in advance. The following people were specially helpful:

- **Frodo Looijaard from the Netherlands** For a host of useful suggestions, and information about the 2.1.x kernels.
- **Stephen Judd from New Zealand** Spelling corrections.
- **Magnus Ahltorp from Sweden** Correcting a mistake of mine about the difference between character and block devices.

0.4.2 For version 1.1.0

- **Emmanuel Papirakis from Quebec, Canada** For porting all of the examples to version 2.2 of the kernel.
- **Frodo Looijaard from the Netherlands** For telling me how to create a multiple file kernel module (1.2).

Of course, any remaining mistakes are my own, and if you think they make the book unusable you're welcome to apply for a full refund of the money you paid me for it.

Chapter 1

Hello, world

When the first caveman programmer chiseled the first program on the walls of the first cave computer, it was a program to paint the string 'Hello, world' in Antelope pictures. Roman programming textbooks began with the 'Salut, Mundi' program. I don't know what happens to people who break with this tradition, and I think it's safer not to find out.

A kernel module has to have at least two functions: `init_module` which is called when the module is inserted into the kernel, and `cleanup_module` which is called just before it is removed. Typically, `init_module` either registers a handler for something with the kernel, or it replaces one of the kernel function with its own code (usually code to do something and then call the original function). The `cleanup_module` function is supposed to undo whatever `init_module` did, so the module can be unloaded safely.

hello.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 Makefiles for Kernel Modules

A kernel module is not an independant executable, but an object file which will be linked into the kernel in runtime. As a result, they should be compiled with the `-c` flag. Also, all kernel modules have to be compiled with certain symbols defined.

- `__KERNEL__` — This tells the header files that this code will be run in kernel mode, not as part of a user process.

- `MODULE` — This tells the header files to give the appropriate definitions for a kernel module.

- `LINUX` — Technically speaking, this is not necessary. However, if you ever want to write a serious kernel module which will compile on more than one operating system, you'll be happy you did. This will allow you to do conditional compilation on the parts which are OS dependant.

There are other symbols which have to be included, or not, depending on the flags the kernel was compiled with. If you're not sure how the kernel was compiled, look it up in `/usr/include/linux/config.h`

- `__SMP__` — Symmetrical MultiProcessing. This has to be defined if the kernel was compiled to support symmetrical multiprocessing (even if it's running just on one CPU). If you use Symmetrical MultiProcessing, there are other things you need to do (see chapter 12).

- `CONFIG_MODVERSIONS` — If CONFIG_MODVERSIONS was enabled, you need to have it defined when compiling the kernel module and and to include `/usr/include/linux/modversions.h`. This can also be done by the code itself.

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 X.
```

So, now the only thing left is to `su` to root (you didn't compile this as root, did you? Living on the edge[1]...), and then `insmod hello` and `rmmod hello` to your heart's content. While you do it, notice your new kernel module in `/proc/modules`.

By the way, the reason why the Makefile recommends against doing `insmod` from X is because when the kernel has a message to print with `printk`, it sends it to the console. When you don't use X, it just goes to the virtual terminal you're using (the one you chose with Alt-F<n>) and you see it. When you do use X, on the other hand, there are two possibilities. Either you have a console open with `xterm -C`, in which case the output will be sent there, or you don't, in which case the output will go to virtual terminal 7 — the one 'covered' by X.

If your kernel becomes unstable you're likelier to get the debug messages without X. Outside of X, `printk` goes directly from the kernel to the console. In X, on the other hand, `printk`'s go to a user mode process (`xterm -C`). When that process receives CPU time, it is supposed to send it to the X server process. Then, when the X server receives the CPU, it is supposed to display it — but an unstable kernel usually means that the system is about to crash or reboot, so you don't want to delay the error messages, which might explain to you what went wrong, for longer than you have to.

1.2 Multiple File Kernel Modules

Sometimes it makes sense to divide a kernel module between several source files. In this case, you need to do the following:

1. In all the source files but one, add the line `#define __NO_VERSION__`. This is important because `module.h` normally includes the definition of `kernel_version`, a global variable with the kernel version the module is compiled for. If you need `version.h`, you need to include it yourself, because `module.h` won't do it for you with `__NO_VERSION__`.

2. Compile all the source files as usual.

3. Combine all the object files into a single one. Under x86, do it with `ld -m elf_i386 -r -o <name`

[1] The reason I prefer not to compile as root is that the least done as root the safer the box is. I work in computer security, so I'm paranoid

of module>.o <1st source file>.o <2nd source file>.o.

Here's an example of such a kernel module.

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;
```

```
}
```

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");
}
```

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
```

Chapter 2

Character Device Files

So, now we're bold kernel programmers and we know how to write kernel modules to do nothing. We feel proud of ourselves and we hold our heads up high. But somehow we get the feeling that something is missing. Catatonic modules are not much fun.

There are two major ways for a kernel module to talk to processes. One is through device files (like the files in the `/dev` directory), the other is to use the proc file system. Since one of the major reasons to write something in the kernel is to support some kind of hardware device, we'll begin with device files.

The original purpose of device files is to allow processes to communicate with device drivers in the kernel, and through them with physical devices (modems, terminals, etc.). The way this is implemented is the following.

Each device driver, which is responsible for some type of hardware, is assigned its own major number. The list of drivers and their major numbers is available in `/proc/devices`. Each physical device managed by a device driver is assigned a minor number. The `/dev` directory is supposed to include a special file, called a device file, for each of those devices, whether or not it's really installed on the system.

For example, if you do `ls -l /dev/hd[ab]*`, you'll see all of the IDE hard disk partitions which might be connected to a machine. Notice that all of them use the same major number, 3, but the minor number changes from one to the other *Disclaimer: This assumes you're using a PC architecture. I don't know about devices on Linux running on other architectures*.

When the system was installed, all of those device files were created by the `mknod` command. There's no technical reason why they have to be in the `/dev` directory, it's just a useful convention. When creating a device file for testing purposes, as with the exercise here, it would probably make more sense to place it in the directory where you compile the kernel module.

Devices are divided into two types: character devices and block devices. The difference is that block devices have a buffer for requests, so they can choose by which order to respond to them. This is important in the case of storage devices, where it's faster to read or write sectors which are close to each other, rather than those which are further apart. Another difference is that block devices can only accept input and return output in blocks (whose size can vary according to the device), whereas character devices are allowed to use as many or as few bytes as they like. Most devices in the world are character, because they don't need this type of buffering, and they don't operate with a fixed block size. You can tell whether a device file is for a block device or a character device by looking at the first character in the output of `ls -l`. If it's 'b' then it's a block device, and if it's 'c' then it's a character device.

This module is divided into two separate parts: The module part which registers the device and the device driver part. The `init_module` function calls `module_register_chrdev` to add the device driver to the kernel's character device driver table. It also returns the major number to be used for the driver. The `cleanup_module` function deregisters the device.

This (registering something and unregistering it) is the general functionality of those two functions. Things in the kernel don't run on their own initiative, like processes, but are called, by processes via system calls, or by hardware devices via interrupts, or by other parts of the kernel (simply by calling specific functions). As a result, when you add code to the kernel, you're supposed to register it as the handler for a certain type of event and when you remove it, you're supposed to unregister it.

The device driver proper is composed of the four device_<action> functions, which are called when somebody tries to do something with a device file which has our major number. The way the kernel knows to call them is via the `file_operations` structure, `Fops`, which was given when the device was registered, which includes pointers to those four functions.

Another point we need to remember here is that we can't allow the kernel module to be `rmmod`ed whenever root feels like it. The reason is that if the device file is opened by a process and then we remove the kernel module, using the file would cause a call to the memory location where the appropriate function (read/write) used to be. If we're lucky, no other code was loaded there, and we'll get an ugly error message. If we're unlucky, another kernel module was loaded into the same location, which means a jump into the middle of another function within the kernel. The results of this would be impossible to predict, but they can't be positive.

Normally, when you don't want to allow something, you return an error code (a negative number) from the function which is supposed to do it. With `cleanup_module` that is impossible because it's a void function. Once `cleanup_module` is called, the module is dead. However, there is a use counter which counts how many other kernel modules are using this kernel module, called the reference count (that's the last number of the line in `/proc/modules`). If this number isn't zero, `rmmod` will fail. The module's reference count is available in the variable `mod_use_count_`. Since there are macros defined for handling this variable (`MOD_INC_USE_COUNT` and `MOD_DEC_USE_COUNT`), we prefer to use them, rather than `mod_use_count_` directly, so we'll be safe if the implementation changes in the future.

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 Multiple Kernel Versions Source Files

The system calls, which are the major interface the kernel shows to the processes, generally stay the same across versions. A new system call may be added, but usually the old ones will behave exactly like they used to. This is necessary for backward compatibility — a new kernel version is **not** supposed to break regular processes. In most cases, the device files will also remain the same. On the other hand, the internal interfaces within the kernel can and do change between versions.

The Linux kernel versions are divided between the stable versions (n.<even number>.m) and the development versions (n.<odd number>.m). The development versions include all the cool new ideas, including those which will be considered a mistake, or reimplemented, in the next version. As a result, you can't trust the interface to remain the same in those versions (which is why I don't bother to support them in this book, it's too much work and it would become dated too quickly). In the stable versions, on the other hand, we can expect the interface to remain the same regardless of the bug fix version (the m number).

This version of the MPG includes support for both version 2.0.x and version 2.2.x of the Linux kernel. Since there are differences between the two, this requires conditional compilation depending on the kernel version. The way to do this to use the macro `LINUX_VERSION_CODE`. In version a.b.c of the kernel, the value of this macro would be $2^{16}a + 2^8 b + c$. To get the value for a specific kernel version, we can use the `KERNEL_VERSION` macro. Since it's not defined in 2.0.35, we define it ourselves if necessary.

Chapter 3

The /proc File System

In Linux there is an additional mechanism for the kernel and kernel modules to send information to processes — the `/proc` file system. Originally designed to allow easy access to information about processes (hence the name), it is now used by every bit of the kernel which has something interesting to report, such as `/proc/modules` which has the list of modules and `/proc/meminfo` which has memory usage statistics.

The method to use the proc file system is very similar to the one used with device drivers — you create a structure with all the information needed for the `/proc` file, including pointers to any handler functions (in our case there is only one, the one called when somebody attempts to read from the `/proc` file). Then, `init_module` registers the structure with the kernel and `cleanup_module` unregisters it.

The reason we use `proc_register_dynamic`[1] is because we don't want to determine the inode number used for our file in advance, but to allow the kernel to determine it to prevent clashes. Normal file systems are located on a disk, rather than just in memory (which is where `/proc` is), and in that case the inode number is a pointer to a disk location where the file's index-node (inode for short) is located. The inode contains information about the file, for example the file's permissions, together with a pointer to the disk location or locations where the file's data can be found.

Because we don't get called when the file is opened or closed, there's no where for us to put `MOD_INC_USE_COUNT` and `MOD_DEC_USE_COUNT` in this module, and if the file is opened and then the module is removed, there's no way to avoid the consequences. In the next chapter we'll see a harder to implement, but more flexible, way of dealing with

[1] In version 2.0, in version 2.2 this is done for us automatically if we set the inode to zero.

/proc files which will allow us to protect against this problem as well.

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);
}
```

Chapter 4

Using /proc For Input

So far we have two ways to generate output from kernel modules: we can register a device driver and mknod a device file, or we can create a /proc file. This allows the kernel module to tell us anything it likes. The only problem is that there is no way for us to talk back. The first way we'll send input to kernel modules will be by writing back to the /proc file.

Because the proc filesystem was written mainly to allow the kernel to report its situation to processes, there are no special provisions for input. The proc_dir_entry struct doesn't include a pointer to an input function, the way it includes a pointer to an output function. Instead, to write into a /proc file, we need to use the standard filesystem mechanism.

In Linux there is a standard mechanism for file system registration. Since every file system has to have its own functions to handle inode and file operations[1], there is a special structure to hold pointers to all those functions, struct inode_operations, which includes a pointer to struct file_operations. In /proc, whenever we register a new file, we're allowed to specify which struct inode_operations will be used for access to it. This is the mechanism we use, a struct inode_operations which includes a pointer to a struct file_operations which includes pointers to our module_input and module_output functions.

It's important to note that the standard roles of read and write are reversed in the kernel. Read functions are used for output, whereas write functions are used for input. The reason

[1] The difference between the two is that file operations deal with the file itself, and inode operations deal with ways of referencing the file, such as creating links to it.

for that is that read and write refer to the user's point of view — if a process reads something from the kernel, then the kernel needs to output it, and if a process writes something to the kernel, then the kernel receives it as input.

Another interesting point here is the `module_permission` function. This function is called whenever a process tries to do something with the `/proc` file, and it can decide whether to allow access or not. Right now it is only based on the operation and the uid of the current used (as available in `current`, a pointer to a structure which includes information on the currently running process), but it could be based on anything we like, such as what other processes are doing with the same file, the time of day, or the last input we received.

The reason for `put_user` and `get_user` is that Linux memory (under Intel architecture, it may be different under some other processors) is segmented. This means that a pointer, by itself, does not reference a unique location in memory, only a location in a memory segment, and you need to know which memory segment it is to be able to use it. There is one memory segment for the kernel, and one of each of the processes.

The only memory segment accessible to a process is its own, so when writing regular programs to run as processes, there's no need to worry about segments. When you write a kernel module, normally you want to access the kernel memory segment, which is handled automatically by the system. However, when the content of a memory buffer needs to be passed between the currently running process and the kernel, the kernel function receives a pointer to the memory buffer which is in the process segment. The `put_user` and `get_user` macros allow you to access that memory.

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);
}
```

Chapter 5

Talking to Device Files (writes and IOCTLs)

Device files are supposed to represent physical devices. Most physical devices are used for output as well as input, so there has to be some mechanism for device drivers in the kernel to get the output to send to the device from processes. This is done by opening the device file for output and writing to it, just like writing to a file. In the following example, this is implemented by `device_write`.

This is not always enough. Imagine you had a serial port connected to a modem (even if you have an internal modem, it is still implemented from the CPU's perspective as a serial port connected to a modem, so you don't have to tax your imagination too hard). The natural thing to do would be to use the device file to write things to the modem (either modem commands or data to be sent through the phone line) and read things from the modem (either responses for commands or the data received through the phone line). However, this leaves open the question of what to do when you need to talk to the serial port itself, for example to send the rate at which data is sent and received.

The answer in Unix is to use a special function called `ioctl` (short for **i**nput **o**utput **c**on**t**ro**l**). Every device can have its own `ioctl` commands, which can be read `ioctl`'s (to send information from a process to the kernel), write `ioctl`'s (to return information to a process), [1] both or neither. The ioctl function is called with three parameters: the file descriptor of the appropriate device file, the ioctl number, and a parameter, which is of type

[1] Notice that here the roles of read and write are reversed *again*, so in `ioctl`'s read is to send information to the kernel and write is to receive information from the kernel.

long so you can use a cast to use it to pass anything. [2]

The ioctl number encodes the major device number, the type of the ioctl, the command, and the type of the parameter. This ioctl number is usually created by a macro call (`_IO`, `_IOR`, `_IOW` or `_IOWR` — depending on the type) in a header file. This header file should then be `#include`'d both by the programs which will use `ioctl` (so they can generate the appropriate `ioctl`'s) and by the kernel module (so it can understand it). In the example below, the header file is `chardev.h` and the program which uses it is `ioctl.c`.

If you want to use `ioctl`'s in your own kernel modules, it is best to receive an official `ioctl` assignment, so if you accidentally get somebody else's `ioctl`'s, or if they get yours, you'll know something is wrong. For more information, consult the kernel source tree at '`Documentation/ioctl-number.txt`'.

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
```

[2] This isn't exact. You won't be able to pass a structure, for example, through an ioctl — but you will be able to pass a pointer to the structure.

```
/* 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);
}
```

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
```

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);
}
```

Chapter 6

Startup Parameters

In many of the previous examples, we had to hard-wire something into the kernel module, such as the file name for `/proc` files or the major device number for the device so we can have `ioctl`'s to it. This goes against the grain of the Unix, and Linux, philosophy which is to write flexible program the user can customize.

The way to tell a program, or a kernel module, something it needs before it can start working is by command line parameters. In the case of kernel modules, we don't get `argc` and `argv` — instead, we get something better. We can define global variables in the kernel module and `insmod` will fill them for us.

In this kernel module, we define two of them: `str1` and `str2`. All you need to do is compile the kernel module and then run `insmod str1=xxx str2=yyy`. When `init_module` is called, `str1` will point to the string '`xxx`' and `str2` to the string '`yyy`'.

In version 2.0 there is no type checking on these arguments[1]. If the first character of `str1` or `str2` is a digit the kernel will fill the variable with the value of the integer, rather than a pointer to the string. If a real life situation you have to check for this.

On the other hand, in version 2.2 you use the macro `MACRO_PARM` to tell `insmod` that you expect a parameters, its name *and its type*. This solves the type problem and allows kernel modules to receive strings which begin with a digit, for example.

param.c

[1] There can't be, since under C the object file only has the location of global variables, not their type. That is why header files are necessary

```
/* 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()
{
}
```

Chapter 7

System Calls

So far, the only thing we've done was to use well defined kernel mechanisms to register /proc files and device handlers. This is fine if you want to do something the kernel programmers thought you'd want, such as write a device driver. But what if you want to do something unusual, to change the behavior of the system in some way? Then, you're mostly on your own.

This is where kernel programming gets dangerous. While writing the example below, I killed the open system call. This meant I couldn't open any files, I couldn't run any programs, and I couldn't shutdown the computer. I had to pull the power switch. Luckily, no files died. To ensure you won't lose any files either, please run sync right before you do the insmod and the rmmod.

Forget about /proc files, forget about device files. They're just minor details. The *real* process to kernel communication mechanism, the one used by all processes, is system calls. When a process requests a service from the kernel (such as opening a file, forking to a new process, or requesting more memory), this is the mechanism used. If you want to change the behaviour of the kernel in interesting ways, this is the place to do it. By the way, if you want to see which system calls a program uses, run `strace <command> <arguments>`.

In general, a process is not supposed to be able to access the kernel. It can't access kernel memory and it can't call kernel functions. The hardware of the CPU enforces this (that's the reason why it's called 'protected mode'). System calls are an exception to this general rule. What happens is that the process fills the registers with the appropriate values and then calls a special instruction which jumps to a previously defined location in the kernel (of course, that location is readable by user processes, it is not writable by them). Under Intel CPUs, this is done by means of interrupt 0x80. The hardware knows that once you jump to this location, you are no longer running in restricted user mode, but as the operating system kernel — and therefore you're allowed to do whatever you want.

The location in the kernel a process can jump to is called `system_call`. The procedure at that location checks the system call number, which tells the kernel what service the process requested. Then, it looks at the table of system calls (`sys_call_table`) to see the address of the kernel function to call. Then it calls the function, and after it returns, does a few system checks and then return back to the process (or to a different process, if the process time ran out). If you want to read this code, it's at the source file `arch/<architecture>/kernel/entry.S`, after the line `ENTRY(system_call)`.

So, if we want to change the way a certain system call works, what we need to do is to write our own function to implement it (usually by adding a bit of our own code, and then calling the original function) and then change the pointer at `sys_call_table` to point to our function. Because we might be removed later and we don't want to leave the system in an unstable state, it's important for `cleanup_module` to restore the table to its original state.

The source code here is an example of such a kernel module. We want to 'spy' on a certain user, and to `printk` a message whenever that user opens a file. Towards this end, we replace the system call to open a file with our own function, called `our_sys_open`. This function checks the uid (user's id) of the current process, and if it's equal to the uid we spy on, it calls `printk` to display the name of the file to be opened. Then, either way, it calls the original `open` function with the same parameters, to actually open the file.

The `init_module` function replaces the appropriate location in `sys_call_table` and keeps the original pointer in a variable. The `cleanup_module` function uses that variable to restore everything back to normal. This approach is dangerous, because of the possibility of two kernel modules changing the same system call. Imagine we have two kernel modules, A and B. A's open system call will be A_open and B's will be B_open. Now, when A is inserted into the kernel, the system call is replaced with A_open, which will call the original sys_open when it's done. Next, B is inserted into the kernel, which replaces the system call with B_open, which will call what it thinks is the original system call, A_open, when it's done.

Now, if B is removed first, everything will be well — it will simply restore the system call to A_open, which calls the original. However, if A is removed and then B is removed, the system will crash. A's removal will restore the system call to the original, sys_open, cutting B out of the loop. Then, when B is removed, it will restore the system call to what **it** thinks is the original, A_open, which is no longer in memory. At first glance, it appears we could solve this particular problem by checking if the system call is equal to our open function and if so not changing it at all (so that B won't change the system call when it's removed), but that will cause an even worse problem. When A is removed, it sees that the system call was changed to B_open so that it is no longer pointing to A_open, so it won't restore it to sys_open before it is removed from memory. Unfortunately, B_open will still try to call A_open which is no longer there, so that even without removing B the system would crash.

I can think of two ways to prevent this problem. The first is to restore the call to the original value, sys_open. Unfortunately, sys_open is not part of the kernel system table in `/proc/ksyms`, so we can't access it. The other solution is to use the reference count to prevent root from `rmmod`'ing the module once it is loaded. This is good for production modules, but bad for an educational sample — which is why I didn't do it here.

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;
}
```

Chapter 8

Blocking Processes

What do you do when somebody asks you for something you can't do right away? If you're a human being and you're bothered by a human being, the only thing you can say is: 'Not right now, I'm busy. *Go away!*'. But if you're a kernel module and you're bothered by a process, you have another possibility. You can put the process to sleep until you can service it. After all, processes are being put to sleep by the kernel and woken up all the time (that's the way multiple processes appear to run on the same time on a single CPU).

This kernel module is an example of this. The file (called `/proc/sleep`) can only be opened by a single process at a time. If the file is already open, the kernel module calls `module_interruptible_sleep_on`[1]. This function changes the status of the task (a task is the kernel data structure which holds information about a process and the system call it's in, if any) to `TASK_INTERRUPTIBLE`, which means that the task will not run until it is woken up somehow, and adds it to `WaitQ`, the queue of tasks waiting to access the file. Then, the function calls the scheduler to context switch to a different process, one which has some use for the CPU.

When a process is done with the file, it closes it, and `module_close` is called. That function wakes up all the processes in the queue (there's no mechanism to only wake up one of them). It then returns and the process which just closed the file can continue to run. In time, the scheduler decides that that process has had enough and gives control of the CPU to another process. Eventually, one of the processes which was in the queue will be given control of the CPU by the scheduler. It starts at the point right after the call to `module_interruptible_sleep_on` [2]. It can then proceed to set a global variable to

[1] The easiest way to keep a file open is to open it with `tail -f`.

[2] This means that the process is still in kernel mode — as far as the process is concerned, it issued the `open`

tell all the other processes that the file is still open and go on with its life. When the other processes get a piece of the CPU, they'll see that global variable and go back to sleep.

To make our life more interesting, `module_close` doesn't have a monopoly on waking up the processes which wait to access the file. A signal, such as Ctrl-C (`SIGINT`) can also wake up a process[3]. In that case, we want to return with `-EINTR` immediately. This is important so users can, for example, kill the process before it receives the file.

There is one more point to remember. Some times processes don't want to sleep, they want either to get what they want immediately, or to be told it cannot be done. Such processes use the `O_NONBLOCK` flag when opening the file. The kernel is supposed to respond by returning with the error code `-EAGAIN` from operations which would otherwise block, such as opening the file in this example. The program cat_noblock, available in the source directory for this chapter, can be used to open a file with `O_NONBLOCK`.

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
```

system call and the system call hasn't returned yet. The process doesn't know somebody else used the CPU for most of the time between the moment it issued the call and the moment it returned.

[3] This is because we used `module_interruptible_sleep_on`. We could have used `module_sleep_on` instead, but that would have resulted is extremely angry users whose control C's are ignored.

```
/* 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);
}
```

Chapter 9

Replacing printk's

In the beginning (chapter 1), I said that X and kernel module programming don't mix. That's true while developing the kernel module, but in actual use you want to be able to send messages to whichever tty[1] the command to the module came from. This is important for identifying errors after the kernel module is released, because it will be used through all of them.

The way this is done is by using `current`, a pointer to the currently running task, to get the current task's tty structure. Then, we look inside that tty structure to find a pointer to a string write function, which we use to write a string to the tty.

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 */
```

[1] **Teletype**, originally a combination keyboard–printer used to communicate with a Unix system, and today an abstraction for the text stream used for a Unix program, whether it's a physical terminal, an xterm on an X display, a network connection used with telnet, etc.

```
/* 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");
}
```

Chapter 10

Scheduling Tasks

Very often, we have 'housekeeping' tasks which have to be done at a certain time, or every so often. If the task is to be done by a process, we do it by putting it in the `crontab` file . If the task is to be done by a kernel module, we have two possibilities. The first is to put a process in the `crontab` file which will wake up the module by a system call when necessary, for example by opening a file. This is terribly inefficient, however — we run a new process off of `crontab`, read a new executable to memory, and all this just to wake up a kernel module which is in memory anyway.

Instead of doing that, we can create a function that will be called once for every timer interrupt. The way we do this is we create a task, held in a `struct tq_struct`, which will hold a pointer to the function. Then, we use `queue_task` to put that task on a task list called `tq_timer`, which is the list of tasks to be executed on the next timer interrupt. Because we want the function to keep on being executed, we need to put it back on `tq_timer` whenever it is called, for the next timer interrupt.

There's one more point we need to remember here. When a module is removed by `rmmod`, first its reference count is checked. If it is zero, `module_cleanup` is called. Then, the module is removed from memory with all its functions. Nobody checks to see if the timer's task list happens to contain a pointer to one of those functions, which will no longer be available. Ages later (from the computer's perspective, from a human perspective it's nothing, less than a hundredth of a second), the kernel has a timer interrupt and tries to call the function on the task list. Unfortunately, the function is no longer there. In most cases, the memory page where it sat is unused, and you get an ugly error message. But if some other code is now sitting at the same memory location, things could get **very** ugly.

Unfortunately, we don't have an easy way to unregister a task from a task list.

Since `cleanup_module` can't return with an error code (it's a void function), the solution is to not let it return at all. Instead, it calls `sleep_on` or `module_sleep_on`[1] to put the `rmmod` process to sleep. Before that, it informs the function called on the timer interrupt to stop attaching itself by setting a global variable. Then, on the next timer interrupt, the `rmmod` process will be woken up, when our function is no longer in the queue and it's safe to remove the module.

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>
```

[1] They're really the same.

```
/* 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);
}
```

Chapter 11

Interrupt Handlers

Except for the last chapter, everything we did in the kernel so far we've done as a response to a process asking for it, either by dealing with a special file, sending an `ioctl`, or issuing a system call. But the job of the kernel isn't just to respond to process requests. Another job, which is every bit as important, is to speak to the hardware connected to the machine.

There are two types of interaction between the CPU and the rest of the computer's hardware. The first type is when the CPU gives orders to the hardware, the other is when the hardware needs to tell the CPU something. The second, called interrupts, is much harder to implement because it has to be dealt with when convenient for the hardware, not the CPU. Hardware devices typically have a very small amount of ram, and if you don't read their information when available, it is lost.

Under Linux, hardware interrupts are called IRQs (short for **I**nterrupt **Req**uests)[1]. There are two types of IRQs, short and long. A short IRQ is one which is expected to take a **very** short period of time, during which the rest of the machine will be blocked and no other interrupts will be handled. A long IRQ is one which can take longer, and during which other interrupts may occur (but not interrupts from the same device). If at all possible, it's better to declare an interrupt handler to be long.

When the CPU receives an interrupt, it stops whatever it's doing (unless it's processing a more important interrupt, in which case it will deal with this one only when the more important one is done), saves certain parameters on the stack and calls the interrupt handler. This means that certain things are not allowed in the interrupt handler itself, because the

[1] This is standard nomencalture on the Intel architecture where Linux originated.

system is in an unknown state. The solution to this problem is for the interrupt handler to do what needs to be done immediately, usually read something from the hardware or send something to the hardware, and then schedule the handling of the new information at a later time (this is called the 'bottom half') and return. The kernel is then guaranteed to call the bottom half as soon as possible — and when it does, everything allowed in kernel modules will be allowed.

The way to implement this is to call `request_irq` to get your interrupt handler called when the relevant IRQ is received (there are 16 of them on Intel platforms). This function receives the IRQ number, the name of the function, flags, a name for `/proc/interrupts` and a parameter to pass to the interrupt handler. The flags can include `SA_SHIRQ` to indicate you're willing to share the IRQ with other interrupt handlers (usually because a number of hardware devices sit on the same IRQ) and `SA_INTERRUPT` to indicate this is a fast interrupt. This function will only succeed if there isn't already a handler on this IRQ, or if you're both willing to share.

Then, from within the interrupt handler, we communicate with the hardware and then use `queue_task_irq` with `tq_immediate` and `mark_bh(BH_IMMEDIATE)` to schedule the bottom half. The reason we can't use the standard `queue_task` in version 2.0 is that the interrupt might happen right in the middle of somebody else's `queue_task`[2]. We need `mark_bh` because earlier versions of Linux only had an array of 32 bottom halves, and now one of them (`BH_IMMEDIATE`) is used for the linked list of bottom halves for drivers which didn't get a bottom half entry assigned to them.

11.1 Keyboards on the Intel Architecture

Warning: The rest of this chapter is completely Intel specific. If you're not running on an Intel platform, it will not work. Don't even try to compile the code here.

I had a problem with writing the sample code for this chapter. On one hand, for an example to be useful it has to run on everybody's computer with meaningful results. On the other hand, the kernel already includes device drivers for all of the common devices, and those device drivers won't coexist with what I'm going to write. The solution I've found was to write something for the keyboard interrupt, and disable the regular keyboard interrupt handler first. Since it is defined as a static symbol in the kernel source files (specifically, `drivers/char/keyboard.c`), there is no way to restore it. Before insmod'ing this code, do on another terminal `sleep 120 ; reboot` if you value your file system.

[2] `queue_task_irq` is protected from this by a global lock — in 2.2 there is no `queue_task_irq` and `queue_task` is protected by a lock.

This code binds itself to IRQ 1, which is the IRQ of the keyboard controlled under Intel architectures. Then, when it receives a keyboard interrupt, it reads the keyboard's status (that's the purpose of the `inb(0x64)`) and the scan code, which is the value returned by the keyboard. Then, as soon as the kernel think it's feasible, it runs `got_char` which gives the code of the key used (the first seven bits of the scan code) and whether it has been pressed (if the 8th bit is zero) or released (if it's one).

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);
}
```

Chapter 12

Symmetrical Multi–Processing

One of the easiest (read, cheapest) ways to improve hardware performance is to put more than one CPU on the board. This can be done either making the different CPUs take on different jobs (asymmetrical multi–processing) or by making them all run in parallel, doing the same job (symmetrical multi–processing, a.k.a. SMP). Doing asymmetrical multi–processing effectively requires specialized knowledge about the tasks the computer should do, which is unavailable in a general purpose operating system such as Linux. On the other hand, symmetrical multi–processing is relatively easy to implement.

By relatively easy, I mean exactly that — not that it's *really* easy. In a symmetrical multi–processing environment, the CPUs share the same memory, and as a result code running in one CPU can affect the memory used by another. You can no longer be certain that a variable you've set to a certain value in the previous line still has that value — the other CPU might have played with it while you weren't looking. Obviously, it's impossible to program like this.

In the case of process programming this normally isn't an issue, because a process will normally only run on one CPU at a time[1]. The kernel, on the other hand, could be called by different processes running on different CPUs.

In version 2.0.x, this isn't a problem because the entire kernel is in one big spinlock. This means that if one CPU is in the kernel and another CPU wants to get in, for example because of a system call, it has to wait until the first CPU is done. This makes Linux SMP safe[2], but terriably inefficient.

[1] The exception is threaded processes, which can run on several CPUs at once.

[2] Meaning it is safe to use it with SMP

In version 2.2.x, several CPUs can be in the kernel at the same time. This is something module writers need to be aware of. I got somebody to give me access to an SMP box, so hopefully the next version of this book will include more information.

Chapter 13

Common Pitfalls

Before I send you on your way to go out into the world and write kernel modules, there are a few things I need to warn you about. If I fail to warn you and something bad happen, please report the problem to me for a full refund of the amount I got paid for your copy of the book.

1. **Using standard libraries** You can't do that. In a kernel module you can only use kernel functions, which are the functions you can see in `/proc/ksyms`.

2. **Disabling interrupts** You might need to do this for a short time and that is OK, but if you don't enable them afterwards, your system will be stuck and you'll have to power it off.

3. **Sticking your head inside a large carnivore** I probably don't have to warn you about this, but I figured I will anyway, just in case.

Appendix A

Changes between 2.0 and 2.2

I don't know the entire kernel well enough do document all of the changes. In the course of converting the examples (or actually, adapting Emmanuel Papirakis's changes) I came across the following differences. I listed all of them here together to help module programmers, especially those who learned from previous versions of this book and are most familiar with the techniques I use, convert to the new version.

An additional resource for people who wish to convert to 2.2 is in `http://www.atnf.csiro.au/~rgooch/linux/docs/porting-to-2.2.html`.

1. **asm/uaccess.h** If you need `put_user` or `get_user` you have to #include it.
2. **get_user** In version 2.2, `get_user` receives both the pointer into user memory and the variable in kernel memory to fill with the information. The reason for this is that `get_user` can now read two or four bytes at a time if the variable we read is two or four bytes long.
3. **file_operations** This structure now has a flush function between the `open` and `close` functions.
4. **close in file_operations** In version 2.2, the close function returns an integer, so it's allowed to fail.
5. **read and write in file_operations** The headers for these functions changed. They now return `ssize_t` instead of an integer, and their parameter list is different. The inode is no longer a parameter, and on the other hand the offset into the file is.

6. **proc_register_dynamic** This function no longer exists. Instead, you call the regular `proc_register` and put zero in the inode field of the structure.

7. **Signals** The signals in the task structure are no longer a 32 bit integer, but an array of `_NSIG_WORDS` integers.

8. **queue_task_irq** Even if you want to scheduale a task to happen from inside an interrupt handler, you use `queue_task`, not `queue_task_irq`.

9. **Module Parameters** You no longer just declare module parameters as global variables. In 2.2 you have to also use `MODULE_PARM` to declare their type. This is a big improvement, because it allows the module to receive string parameters which start with a digits, for example, without getting confused.

10. **Symmetrical Multi–Processing** The kernel is no longer inside one huge spinlock, which means that kernel modules have to be aware of SMP.

Appendix B

Where From Here?

I could easily have squeezed a few more chapters into this book. I could have added a chapter about creating new file systems, or about adding new protocols stacks (as if there's a need for that — you'd have to dig under ground to find a protocol stack not supported by Linux). I could have added explanations of the kernel mechanisms we haven't touched upon, such as bootstrapping or the disk interface.

However, I chose not to. My purpose in writing this book was to provide initiation into the mysteries of kernel module programming and to teach the common techniques for that purpose. For people seriously interested in kernel programming, I recommend the list of kernel resources in `http://jungla.dit.upm.es/~jmseyas/linux/kernel/hackers-docs.html`. Also, as Linus said, the best way is to learn the kernel is to read the source code yourself.

If you're interested in more examples of short kernel modules, I recommend Phrack magazine. Even if you're not interested in security, and as a programmer you should be, the kernel modules there are good examples of what you can do inside the kernel, and they're short enough not to require too much effort to understand.

I hope I have helped you in your quest to become a better programmer, or at least to have fun through technology. And, if you do write useful kernel modules, I hope you publish them under the GPL, so I can use them too.

Appendix C

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.

C.1 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.

Appendix D

Showing Your Appreciation

This is a free document. You have no obligations beyond those given in the GNU Public License (Appendix E). However, if you want to do something in return for getting this book, there are a few things you could do.

- Send me a postcard to
  ```
  Ori Pomerantz
  Apt.  #1032
  2355 N Hwy 360
  Grand Prairie
  TX 75050
  USA
  ```
  If you want to receive a thank-you from me, include your e-mail address.

- Contribute money, or better yet, time, to the free software community. Write a program or a document and publish it under the GPL. Teach other people how to use free software, such as Linux or Perl.

- Explain to people how being selfish is *not* incompatible with living in a society or with helping other people. I enjoyed writing this document, and I believe publishing it will contribute to me in the future. At the same time, I wrote a book which, if you got this far, helps you. Remember that happy people are usually more useful to oneself than unhappy people, and able people are *way* better than people of low ability.

- **Be happy**. If I get to meet you, it will make the encounter better for me, it will make you more useful for me ;-).

Appendix E

The GNU General Public License

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

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

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:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for
details type show w.  This is free software, and you are
welcome to redistribute it under certain conditions; type
show c for details.
```

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.

Index