The Linux Kernel Module Programming Guide

Peter Jay Salzman
Michael Burian
Ori Pomerantz

Copyright © 2001 Peter Jay Salzman

2004−05−16 ver 2.6.0

The Linux Kernel Module Programming Guide is a free book; you may reproduce and/or modify it under the terms of the Open Software License, version 1.1. You can obtain a copy of this license at http://opensource.org/licenses/osl.php.

This book is distributed in the hope it will be useful, but without any warranty, without even the implied warranty of merchantability or fitness for a particular purpose.

The author encourages wide distribution of this book for personal or commercial use, provided the above copyright notice remains intact and the method adheres to the provisions of the Open Software License. In summary, you may copy and distribute this book free of charge or for a profit. No explicit permission is required from the author for reproduction of this book in any medium, physical or electronic.

Derivative works and translations of this document must be placed under the Open Software License, and the original copyright notice must remain intact. If you have contributed new material to this book, you must make the material and source code available for your revisions. Please make revisions and updates available directly to the document maintainer, Peter Jay Salzman <p@dirac.org>. This will allow for the merging of updates and provide consistent revisions to the Linux community.

If you publish or distribute this book commercially, donations, royalties, and/or printed copies are greatly appreciated by the author and the Linux Documentation Project (LDP). Contributing in this way shows your support for free software and the LDP. If you have questions or comments, please contact the address above.

Table of Contents

Foreword....1
1. Authorship....1
2. Versioning and Notes....1
3. Acknowledgements....1

Chapter 1. Introduction....2
1.1. What Is A Kernel Module?....2
1.2. How Do Modules Get Into The Kernel?....2
1.2.1. Before We Begin....3

Chapter 2. Hello World....5
2.1. Hello, World (part 1): The Simplest Module....5
2.1.1. Introducing printk()....6
2.2. Compiling Kernel Modules....6
2.3. Hello World (part 2)....7
2.4. Hello World (part 3): The __init and __exit Macros....8
2.5. Hello World (part 4): Licensing and Module Documentation....9
2.6. Passing Command Line Arguments to a Module....10
2.7. Modules Spanning Multiple Files....12
2.8. Building modules for a precompiled kernel....14

Chapter 3. Preliminaries....16
3.1. Modules vs Programs....16
3.1.1. How modules begin and end....16
3.1.2. Functions available to modules....16
3.1.3. User Space vs Kernel Space....17
3.1.4. Name Space....17
3.1.5. Code space....18
3.1.6. Device Drivers....18

Chapter 4. Character Device Files....21
4.1. Character Device Drivers....21
4.1.1. The file_operations Structure....21
4.1.2. The file structure....22
4.1.3. Registering A Device....22
4.1.4. Unregistering A Device....23
4.1.5. chardev.c....23
4.1.6. Writing Modules for Multiple Kernel Versions....26

Chapter 5. The /proc File System....28
5.1. The /proc File System....28

Chapter 6. Using /proc For Input....31
6.1. Using /proc For Input....31

Chapter 7. Talking To Device Files....35
7.1. Talking to Device Files (writes and IOCTLs)}....35

Table of Contents

Chapter 8. System Calls....44
8.1. System Calls....44

Chapter 9. Blocking Processes....49
9.1. Blocking Processes....49
9.1.1. Enter Sandman....49

Chapter 10. Replacing Printks....55
10.1. Replacing printk....55
10.2. Flashing keyboard LEDs....56

Chapter 11. Scheduling Tasks....59
11.1. Scheduling Tasks....59

Chapter 12. Interrupt Handlers....63
12.1. Interrupt Handlers....63
12.1.1. Interrupt Handlers....63
12.1.2. Keyboards on the Intel Architecture....64

Chapter 13. Symmetric Multi Processing....67
13.1. Symmetrical Multi−Processing....67

Chapter 14. Common Pitfalls....68
14.1. Common Pitfalls....68

Appendix A. Changes: 2.0 To 2.2....69
A.1. Changes between 2.0 and 2.2....69
A.1.1. Changes between 2.0 and 2.2....69

Appendix B. Where To Go From Here....70
B.1. Where From Here?....70

Foreword

1. Authorship

The Linux Kernel Module Programming Guide was originally written for the 2.2 kernels by Ori Pomerantz. Eventually, Ori no longer had time to maintain the document. After all, the Linux kernel is a fast moving target. Peter Jay Salzman took over maintenance and updated it for the 2.4 kernels. Eventually, Peter no longer had time to follow developments with the 2.6 kernel, so Michael Burian became a co−maintainer to update the document for the 2.6 kernels.

2. Versioning and Notes

The Linux kernel is a moving target. There has always been a question whether the LKMPG should remove deprecated information or keep it around for historical sake. Michael Burian and I decided to create a new branch of the LKMPG for each new stable kernel version. So version LKMPG 2.4.x will address Linux kernel 2.4 and LKMPG 2.6.x will address Linux kernel 2.6. No attempt will be made to archive historical information; a person wishing this information should read the appropriately versioned LKMPG.

The source code and discussions should apply to most architectures, but I can't promise anything. One exception is Chapter 12, Interrupt Handlers, which should not work on any architecture except for x86.

3. Acknowledgements

Ori Pomerantz would like to thank Yoav Weiss for many helpful ideas, discussions, and corrections. He would also like to thank Frodo Looijaard from the Netherlands, Stephen Judd from New Zealand, Magnus Ahltorp from Sweeden and Emmanuel Papirakis from Quebec, Canada.

Peter would also like to thank Ori for letting him take over the LKMPG. He would also like to thank Jeff Newmiller, Rhonda Frances Bailey (who is now Rhonda Frances Salzman) and Mark Kim for teaching him with patience and friendship regardless how busy they were. He would also like to thank David Porter who had the unenviable job of helping convert the original LaTeX source into docbook. It was a long and boring job, but had to be done.

Thanks also goes to the fine people at www.kernelnewbies.org. In particular, Mark McLoughlin and John Levon who I'm sure have much better things to do than to hang out on kernelnewbies.org and teach the newbies. If this guide teaches you anything, they are partially to blame.

Both Ori and I would like to thank Richard M. Stallman and Linus Torvalds for giving us the opportunity to not only run a high−quality operating system, but to take a close peek at how it works.

The following people have contributed corrections or good suggestions: Ignacio Martin, David Porter, Daniele Paolo Scarpazza and Dimo Velev

Chapter 1. Introduction

1.1. What Is A Kernel Module?

So, you want to write a kernel module. You know C, you've written a few 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.

What exactly is a kernel module? Modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality.

1.2. How Do Modules Get Into The Kernel?

You can see what modules are already loaded into the kernel by running **lsmod**, which gets its information by reading the file `/proc/modules`.

How do these modules find their way into the kernel? When the kernel needs a feature that is not resident in the kernel, the kernel module daemon kmod[1] execs modprobe to load the module in. modprobe is passed a string in one of two forms:

- A module name like `softdog` or `ppp`.
- A more generic identifier like `char-major-10-30`.

If modprobe is handed a generic identifier, it first looks for that string in the file `/etc/modules.conf`. If it finds an alias line like:

```
alias char-major-10-30 softdog
```

it knows that the generic identifier refers to the module `softdog.o`.

Next, modprobe looks through the file `/lib/modules/version/modules.dep`, to see if other modules must be loaded before the requested module may be loaded. This file is created by **depmod -a** and contains module dependencies. For example, `msdos.o` requires the `fat.o` module to be already loaded into the kernel. The requested module has a dependancy on another module if the other module defines symbols (variables or functions) that the requested module uses.

Lastly, modprobe uses insmod to first load any prerequisite modules into the kernel, and then the requested module. modprobe directs insmod to `/lib/modules/version/`[2], the standard directory for modules. insmod is intended to be fairly dumb about the location of modules, whereas modprobe is aware of the default location of modules. So for example, if you wanted to load the msdos module, you'd have to either run:

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

or just run "**modprobe −a msdos**".

Linux distros provide modprobe, insmod and depmod as a package called modutils or mod−utils.

Before finishing this chapter, let's take a quick look at a piece of `/etc/modules.conf`:

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

Lines beginning with a '#' are comments. Blank lines are ignored.

The `path[misc]` line tells modprobe to replace the search path for misc modules with the directory `/lib/modules/2.4.?/local`. As you can see, shell meta characters are honored.

The `path[net]` line tells modprobe to look for net modules in the directory `~p/mymodules`, however, the "keep" directive preceding the `path[net]` directive tells modprobe to add this directory to the standard search path of net modules as opposed to replacing the standard search path, as we did for the misc modules.

The alias line says to load in `eepro.o` whenever kmod requests that the generic identifier `eth0' be loaded.

You won't see lines like "alias block−major−2 floppy" in `/etc/modules.conf` because modprobe already knows about the standard drivers which will be used on most systems.

Now you know how modules get into the kernel. There's a bit more to the story if you want to write your own modules which depend on other modules (we calling this `stacking modules'). But this will have to wait for a future chapter. We have a lot to cover before addressing this relatively high−level issue.

1.2.1. Before We Begin

Before we delve into code, there are a few issues we need to cover. Everyone's system is different and everyone has their own groove. Getting your first "hello world" program to compile and load correctly can sometimes be a trick. Rest assured, after you get over the initial hurdle of doing it for the first time, it will be smooth sailing thereafter.

1.2.1.1. Modversioning

A module compiled for one kernel won't load if you boot a different kernel unless you enable `CONFIG_MODVERSIONS` in the kernel. We won't go into module versioning until later in this guide. Until we cover modversions, the examples in the guide may not work if you're running a kernel with modversioning turned on. However, most stock Linux distro kernels come with it turned on. If you're having trouble loading the modules because of versioning errors, compile a kernel with modversioning turned off.

1.2.1.2. Using X

It is highly recommended that you type in, compile and load all the examples this guide discusses. It's also highly recommended you do this from a console. You should not be working on this stuff in X.

Modules can't print to the screen like `printf()` can, but they can log information and warnings, which ends up being printed on your screen, but only on a console. If you insmod a module from an xterm, the information and warnings will be logged, but only to your log files. You won't see it unless you look through your log files. To have immediate access to this information, do all your work from console.

1.2.1.3. Compiling Issues and Kernel Version

Very often, Linux distros will distribute kernel source that has been patched in various non−standard ways, which may cause trouble.

A more common problem is that some Linux distros distribute incomplete kernel headers. You'll need to compile your code using various header files from the Linux kernel. Murphy's Law states that the headers that are missing are exactly the ones that you'll need for your module work.

To avoid these two problems, I highly recommend that you download, compile and boot into a fresh, stock Linux kernel which can be downloaded from any of the Linux kernel mirror sites. See the Linux Kernel HOWTO for more details.

Ironically, this can also cause a problem. By default, gcc on your system may look for the kernel headers in their default location rather than where you installed the new copy of the kernel (usually in `/usr/src/`. This can be fixed by using gcc's `−I` switch.

Chapter 2. Hello World

2.1. Hello, World (part 1): The Simplest Module

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, but I think it's safer not to find out. We'll start with a series of hello world programs that demonstrate the different aspects of the basics of writing a kernel module.

Here's the simplest module possible. Don't compile it yet; we'll cover module compilation in the next section.

Example 2–1. hello–1.c

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

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

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

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

Kernel modules must have at least two functions: a "start" (initialization) function called `init_module()` which is called when the module is insmoded into the kernel, and an "end" (cleanup) function called `cleanup_module()` which is called just before it is rmmoded. Actually, things have changed starting with kernel 2.3.13. You can now use whatever name you like for the start and end functions of a module, and you'll learn how to do this in Section 2.3. In fact, the new method is the preferred method. However, many people still use `init_module()` and `cleanup_module()` for their start and end functions.

Typically, `init_module()` either registers a handler for something with the kernel, or it replaces one of the kernel functions 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.

Lastly, every kernel module needs to include `linux/module.h`. We needed to include `linux/kernel.h` only for the macro expansion for the `printk()` log level, `KERN_ALERT`, which you'll learn about in Section 2.1.1.

2.1.1. Introducing `printk()`

Despite what you might think, `printk()` was not meant to communicate information to the user, even though we used it for exactly this purpose in hello−1! It happens to be a logging mechanism for the kernel, and is used to log information or give warnings. Therefore, each `printk()` statement comes with a priority, which is the `<1>` and `KERN_ALERT` you see. There are 8 priorities and the kernel has macros for them, so you don't have to use cryptic numbers, and you can view them (and their meanings) in `linux/kernel.h`. If you don't specify a priority level, the default priority, `DEFAULT_MESSAGE_LOGLEVEL`, will be used.

Take time to read through the priority macros. The header file also describes what each priority means. In practise, don't use number, like `<4>`. Always use the macro, like `KERN_WARNING`.

If the priority is less than `int console_loglevel`, the message is printed on your current terminal. If both **syslogd** and klogd are running, then the message will also get appended to `/var/log/messages`, whether it got printed to the console or not. We use a high priority, like `KERN_ALERT`, to make sure the `printk()` messages get printed to your console rather than just logged to your logfile. When you write real modules, you'll want to use priorities that are meaningful for the situation at hand.

2.2. Compiling Kernel Modules

Kernel modules need to be compiled with certain gcc options to make them work. In addition, they also need to be compiled with certain symbols defined. Former kernel versions required us to care much about these settings, which are usually stored in Makefiles. Although hierarchically organized, many redundant settings accumulated in sublevel Makefiles and made them large and rather difficult to maintain. Fortunately, there is a new way of doing these things, called kbuild, and the build process for external loadable modules is now fully integrated into the standard kernel build mechanism. To learn more on how to compile modules which are not part of the official kernel (as ours), see file `linux/Documentation/kbuild/modules.txt`.

So, let's look at a simple Makefile for compiling a module named `hello−1.c`:

Example 2–2. Makefile for a basic kernel module

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

Now you can compile the module by issuing the command **make −C /usr/src/linux−`uname −r` SUBDIRS=$PWD modules** . You should obtain an output which resembles the following:

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

Please note that kernel 2.6 introduces a new file naming convention: kernel modules now have a `.ko` extension (in place of the old `.o` extension) which easily distinguishes them from conventional object files. Additional details about Makefiles for kernel modules are available in

`linux/Documentation/kbuild/makefiles.txt`. Be sure to read this and the related files before starting to dig into Makefiles.

Now it is time to insert your freshly−compiled module it into the kernel with **insmod ./hello−1.ko** (ignore anything you see about tainted kernels; we'll cover that shortly).

All modules loaded into the kernel are listed in `/proc/modules`. Go ahead and cat that file to see that your module is really a part of the kernel. Congratulations, you are now the author of Linux kernel code! When the novelty wares off, remove your module from the kernel by using **rmmod hello−1**. Take a look at `/var/log/messages` just to see that it got logged to your system logfile.

Here's another exercise to the reader. See that comment above the return statement in `init_module()`? Change the return value to something non−zero, recompile and load the module again. What happens?

2.3. Hello World (part 2)

As of Linux 2.4, you can rename the init and cleanup functions of your modules; they no longer have to be called `init_module()` and `cleanup_module()` respectively. This is done with the `module_init()` and `module_exit()` macros. These macros are defined in `linux/init.h`. The only caveat is that your init and cleanup functions must be defined before calling the macros, otherwise you'll get compilation errors. Here's an example of this technique:

Example 2−3. hello−2.c

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

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

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

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

So now we have two real kernel modules under our belt. Adding another module is as simple as this:

Example 2−4. Makefile for both our modules

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

Now have a look at `linux/drivers/char/Makefile` for a real world example. As you can see, some things get hardwired into the kernel (obj−y) but where are all those obj−m gone? Those familiar with shell scripts will easily be able to spot them. For those not, the obj−$(CONFIG_FOO) entries you see everywhere expand into obj−y or obj−m, depending on whether the CONFIG_FOO variable has been set to y or m. While we are at it, those were exactly the kind of variables that you have set in the `linux/.config` file, the last time when you said **make menuconfig** or something like that.

2.4. Hello World (part 3): The `__init` and `__exit` Macros

This demonstrates a feature of kernel 2.2 and later. Notice the change in the definitions of the init and cleanup functions. The `__init` macro causes the init function to be discarded and its memory freed once the init function finishes for built−in drivers, but not loadable modules. If you think about when the init function is invoked, this makes perfect sense.

There is also an `__initdata` which works similarly to `__init` but for init variables rather than functions.

The `__exit` macro causes the omission of the function when the module is built into the kernel, and like `__exit`, has no effect for loadable modules. Again, if you consider when the cleanup function runs, this makes complete sense; built−in drivers don't need a cleanup function, while loadable modules do.

These macros are defined in `linux/init.h` and serve to free up kernel memory. When you boot your kernel and see something like `Freeing unused kernel memory: 236k freed`, this is precisely what the kernel is freeing.

Example 2−5. hello−3.c

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

static int hello3_data __initdata = 3;

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

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

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

2.5. Hello World (part 4): Licensing and Module Documentation

If you're running kernel 2.4 or later, you might have noticed something like this when you loaded the previous example modules:

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

In kernel 2.4 and later, a mechanism was devised to identify code licensed under the GPL (and friends) so people can be warned that the code is non open−source. This is accomplished by the `MODULE_LICENSE()` macro which is demonstrated in the next piece of code. By setting the license to GPL, you can keep the warning from being printed. This license mechanism is defined and documented in `linux/module.h`:

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

Similarly, `MODULE_DESCRIPTION()` is used to describe what the module does, `MODULE_AUTHOR()` declares the module's author, and `MODULE_SUPPORTED_DEVICE()` declares what types of devices the module supports.

These macros are all defined in `linux/module.h` and aren't used by the kernel itself. They're simply for documentation and can be viewed by a tool like objdump. As an exercise to the reader, try grepping through `linux/drivers` to see how module authors use these macros to document their modules.

Example 2−6. hello−4.c

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

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

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

module_init(init_hello_4);
module_exit(cleanup_hello_4);

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

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

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

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

2.6. Passing Command Line Arguments to a Module

Modules can take command line arguments, but not with the `argc`/`argv` you might be used to.

To allow arguments to be passed to your module, declare the variables that will take the values of the command line arguments as global and then use the `MODULE_PARM()` macro, (defined in `linux/module.h`) to set the mechanism up. At runtime, insmod will fill the variables with any command line arguments that are given, like **./insmod mymodule.o myvariable=5**. The variable declarations and macros should be placed at the beginning of the module for clarity. The example code should clear up my admittedly lousy explanation.

The `MODULE_PARM()` macro takes 2 arguments: the name of the variable and its type. The supported variable types are "`b`": single byte, "`h`": short int, "`i`": integer, "`l`": long int and "`s`": string, and the integer

types can be signed as usual or unsigned. Strings should be declared as "char *" and insmod will allocate memory for them. You should always try to give the variables an initial default value. This is kernel code, and you should program defensively. For example:

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

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

Arrays are supported too. An integer value preceding the type in MODULE_PARM will indicate an array of some maximum length. Two numbers separated by a '–' will give the minimum and maximum number of values. For example, an array of shorts with at least 2 and no more than 4 values could be declared as:

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

A good use for this is to have the module variable's default values set, like an port or IO address. If the variables contain the default values, then perform autodetection (explained elsewhere). Otherwise, keep the current value. This will be made clear later on.

Lastly, there's a macro function, `MODULE_PARM_DESC()`, that is used to document arguments that the module can take. It takes two parameters: a variable name and a free form string describing that variable.

Example 2–7. hello–5.c

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

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

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

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

module_param(myshort, short, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(myshort, "A short integer");
module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
```

```
MODULE_PARM_DESC(myint, "An integer");
module_param(mylong, long, S_IRUSR);
MODULE_PARM_DESC(mylong, "A long integer");
module_param(mystring, charp, 0000);
MODULE_PARM_DESC(mystring, "A character string");

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

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

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

I would recommend playing around with this code:

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

satan# rmmod hello-5
Goodbye, world 5

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

satan# rmmod hello-5
Goodbye, world 5

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

2.7. Modules Spanning Multiple Files

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

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, use **ld –m elf_i386 –r –o <module name.o> <1st src file.o> <2nd src file.o>**.

The makefile will, once again, save us from having to get our hands dirty with compiling and linking the object files.

Here's an example of such a kernel module.

Example 2–8. start.c

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

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

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

The next file:

Example 2–9. stop.c

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

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

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

And finally, the makefile:

Example 2–10. Makefile

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

2.8. Building modules for a precompiled kernel

Obviously, we strongly suggest you to recompile your kernel, so that you can enable a number of useful debugging features, such as forced module unloading (`MODULE_FORCE_UNLOAD`): when this option is enabled, you can force the kernel to unload a module even when it believes it is unsafe, via a **rmmod –f module** command. This option can save you a lot of time and a number of reboots during the development of a module.

Nevertheless, there is a number of cases in which you may want to load your module into a precompiled running kernel, such as the ones shipped with common Linux distributions, or a kernel you have compiled in the past. In certain circumstances you could require to compile and insert a module into a running kernel which you are not allowed to recompile, or on a machine that you prefer not to reboot. If you can't think of a case that will force you to use modules for a precompiled kernel you might want to skip this and treat the rest of this chapter as a big footnote.

Now, if you just install a kernel source tree, use it to compile your kernel module and you try to insert your module into the kernel, in most cases you would obtain an error as follows:

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

Less cryptical information are logged to `/var/log/messages`:

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

In other words, your kernel refuses to accept your module because version strings (more precisely, version magics) do not match. Incidentally, version magics are stored in the module object in the form of a static string, starting with `vermagic:`. Version data are inserted in your module when it is linked against the `init/vermagic.o` file. To inspect version magics and other strings stored in a given module, issue the **modinfo module.ko** command:

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

To overcome this problem we could resort to the **––force–vermagic** option, but this solution is potentially unsafe, and unquestionably inacceptable in production modules. Consequently, we want to compile our module in an environment which was identical to the one in which our precompiled kernel was built. How to do this, is the subject of the remainder of this chapter.

First of all, make sure that a kernel source tree is available, having exactly the same version as your current kernel. Then, find the configuration file which was used to compile your precompiled kernel. Usually, this is available in your current `/boot` directory, under a name like `config-2.6.x`. You may just want to copy it to your kernel source tree: **cp /boot/config–`uname –r` /usr/src/linux–`uname –r`/.config**.

Let's focus again on the previous error message: a closer look at the version magic strings suggests that, even with two configuration files which are exactly the same, a slight difference in the version magic could be possible, and it is sufficient to prevent insertion of the module into the kernel. That slight difference, namely the `custom` string which appears in the module's version magic and not in the kernel's one, is due to a modification with respect to the original, in the makefile that some distribution include. Then, examine your `/usr/src/linux/Makefile`, and make sure that the specified version information matches exactly the one used for your current kernel. For example, you makefile could start as follows:

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

In this case, you need to restore the value of symbol `EXTRAVERSION` to `-1.358`. We suggest to keep a backup copy of the makefile used to compile your kernel available in `/lib/modules/2.6.5-1.358/build`. A simple **cp /lib/modules/`uname -r`/build/Makefile /usr/src/linux-`uname -r`** should suffice. Additionally, if you already started a kernel build with the previous (wrong) `Makefile`, you should also rerun **make**, or directly modify symbol `UTS_RELEASE` in file `/usr/src/linux-2.6.x/include/linux/version.h` according to contents of file `/lib/modules/2.6.x/build/include/linux/version.h`, or overwrite the latter with the first.

Now, please run **make** to update configuration and version headers and objects:

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

If you do not desire to actually compile the kernel, you can interrupt the build process (**CTRL-C**) just after the `SPLIT` line, because at that time, the files you need will be are ready. Now you can turn back to the directory of your module and compile it: It will be built exactly according your current kernel settings, and it will load into it without any errors.

Chapter 3. Preliminaries

3.1. Modules vs Programs

3.1.1. How modules begin and end

A program usually begins with a `main()` function, executes a bunch of instructions and terminates upon completion of those instructions. Kernel modules work a bit differently. A module always begin with either the `init_module` or the function you specify with `module_init` call. This is the entry function for modules; it tells the kernel what functionality the module provides and sets up the kernel to run the module's functions when they're needed. Once it does this, entry function returns and the module does nothing until the kernel wants to do something with the code that the module provides.

All modules end by calling either `cleanup_module` or the function you specify with the `module_exit` call. This is the exit function for modules; it undoes whatever entry function did. It unregisters the functionality that the entry function registered.

Every module must have an entry function and an exit function. Since there's more than one way to specify entry and exit functions, I'll try my best to use the terms `entry function' and `exit function', but if I slip and simply refer to them as `init_module` and `cleanup_module`, I think you'll know what I mean.

3.1.2. Functions available to modules

Programmers use functions they don't define all the time. A prime example of this is `printf()`. You use these library functions which are provided by the standard C library, libc. The definitions for these functions don't actually enter your program until the linking stage, which insures that the code (for `printf()` for example) is available, and fixes the call instruction to point to that code.

Kernel modules are different here, too. In the hello world example, you might have noticed that we used a function, `printk()` but didn't include a standard I/O library. That's because modules are object files whose symbols get resolved upon insmod'ing. The definition for the symbols comes from the kernel itself; the only external functions you can use are the ones provided by the kernel. If you're curious about what symbols have been exported by your kernel, take a look at `/proc/kallsyms`.

One point to keep in mind is the difference between library functions and system calls. Library functions are higher level, run completely in user space and provide a more convenient interface for the programmer to the functions that do the real work---system calls. System calls run in kernel mode on the user's behalf and are provided by the kernel itself. The library function `printf()` may look like a very general printing function, but all it really does is format the data into strings and write the string data using the low-level system call `write()`, which then sends the data to standard output.

Would you like to see what system calls are made by `printf()`? It's easy! Compile the following program:

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

with **gcc –Wall –o hello hello.c**. Run the exectable with **strace hello**. Are you impressed? Every line you see corresponds to a system call. strace[3] is a handy program that gives you details about what system calls a program is making, including which call is made, what its arguments are what it returns. It's an invaluable tool for figuring out things like what files a program is trying to access. Towards the end, you'll see a line which looks like `write(1, "hello", 5hello)`. There it is. The face behind the `printf()` mask. You may not be familiar with write, since most people use library functions for file I/O (like fopen, fputs, fclose). If that's the case, try looking at **man 2 write**. The 2nd man section is devoted to system calls (like `kill()` and `read()`. The 3rd man section is devoted to library calls, which you would probably be more familiar with (like `cosh()` and `random()`).

You can even write modules to replace the kernel's system calls, which we'll do shortly. Crackers often make use of this sort of thing for backdoors or trojans, but you can write your own modules to do more benign things, like have the kernel write *Tee hee, that tickles!* everytime someone tries to delete a file on your system.

3.1.3. User Space vs Kernel Space

A kernel is all about access to resources, whether the resource in question happens to be a video card, a hard drive or even memory. Programs often compete for the same resource. As I just saved this document, updatedb started updating the locate database. My vim session and updatedb are both using the hard drive concurrently. The kernel needs to keep things orderly, and not give users access to resources whenever they feel like it. To this end, a CPU can run in different modes. Each mode gives a different level of freedom to do what you want on the system. The Intel 80386 architecture has 4 of these modes, which are called rings. Unix uses only two rings; the highest ring (ring 0, also known as `supervisor mode' where everything is allowed to happen) and the lowest ring, which is called `user mode'.

Recall the discussion about library functions vs system calls. Typically, you use a library function in user mode. The library function calls one or more system calls, and these system calls execute on the library function's behalf, but do so in supervisor mode since they are part of the kernel itself. Once the system call completes its task, it returns and execution gets transfered back to user mode.

3.1.4. Name Space

When you write a small C program, you use variables which are convenient and make sense to the reader. If, on the other hand, you're writing routines which will be part of a bigger problem, any global variables you have are part of a community of other peoples' global variables; some of the variable names can clash. When a program has lots of global variables which aren't meaningful enough to be distinguished, you get *namespace pollution*. In large projects, effort must be made to remember reserved names, and to find ways to develop a scheme for naming unique variable names and symbols.

When writing kernel code, even the smallest module will be linked against the entire kernel, so this is definitely an issue. The best way to deal with this is to declare all your variables as static and to use a well–defined prefix for your symbols. By convention, all kernel prefixes are lowercase. If you don't want to declare everything as static, another option is to declare a `symbol table` and register it with a kernel. We'll get to this later.

The file `/proc/kallsyms` holds all the symbols that the kernel knows about and which are therefore accessible to your modules since they share the kernel's codespace.

3.1.5. Code space

Memory management is a very complicated subject−−−the majority of O'Reilly's `Understanding The Linux Kernel' is just on memory management! We're not setting out to be experts on memory managements, but we do need to know a couple of facts to even begin worrying about writing real modules.

If you haven't thought about what a segfault really means, you may be surprised to hear that pointers don't actually point to memory locations. Not real ones, anyway. When a process is created, the kernel sets aside a portion of real physical memory and hands it to the process to use for its executing code, variables, stack, heap and other things which a computer scientist would know about[4]. This memory begins with 0 and extends up to whatever it needs to be. Since the memory space for any two processes don't overlap, every process that can access a memory address, say `0xbffff978`, would be accessing a different location in real physical memory! The processes would be accessing an index named `0xbffff978` which points to some kind of offset into the region of memory set aside for that particular process. For the most part, a process like our Hello, World program can't access the space of another process, although there are ways which we'll talk about later.

The kernel has its own space of memory as well. Since a module is code which can be dynamically inserted and removed in the kernel (as opposed to a semi−autonomous object), it shares the kernel's codespace rather than having its own. Therefore, if your module segfaults, the kernel segfaults. And if you start writing over data because of an off−by−one error, then you're trampling on kernel code. This is even worse than it sounds, so try your best to be careful.

By the way, I would like to point out that the above discussion is true for any operating system which uses a monolithic kernel[5]. There are things called microkernels which have modules which get their own codespace. The GNU Hurd and QNX Neutrino are two examples of a microkernel.

3.1.6. Device Drivers

One class of module is the device driver, which provides functionality for hardware like a TV card or a serial port. On unix, each piece of hardware is represented by a file located in `/dev` named a `device file` which provides the means to communicate with the hardware. The device driver provides the communication on behalf of a user program. So the `es1370.o` sound card device driver might connect the `/dev/sound` device file to the Ensoniq IS1370 sound card. A userspace program like mp3blaster can use `/dev/sound` without ever knowing what kind of sound card is installed.

3.1.6.1. Major and Minor Numbers

Let's look at some device files. Here are device files which represent the first three partitions on the primary master IDE hard drive:

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

Notice the column of numbers separated by a comma? The first number is called the device's major number. The second number is the minor number. The major number tells you which driver is used to access the hardware. Each driver is assigned a unique major number; all device files with the same major number are controlled by the same driver. All the above major numbers are 3, because they're all controlled by the same

driver.

The minor number is used by the driver to distinguish between the various hardware it controls. Returning to the example above, although all three devices are handled by the same driver they have unique minor numbers because the driver sees them as being different pieces of hardware.

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 the best order in which to respond to the requests. 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. The devices you see above are block devices. Here are some character devices (the serial ports):

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

If you want to see which major numbers have been assigned, you can look at `/usr/src/linux/Documentation/devices.txt`.

When the system was installed, all of those device files were created by the **mknod** command. To create a new char device named `coffee' with major/minor number `12` and `2`, simply do **mknod /dev/coffee c 12 2**. You don't *have* to put your device files into `/dev`, but it's done by convention. Linus put his device files in `/dev`, and so should you. However, when creating a device file for testing purposes, it's probably OK to place it in your working directory where you compile the kernel module. Just be sure to put it in the right place when you're done writing the device driver.

I would like to make a few last points which are implicit from the above discussion, but I'd like to make them explicit just in case. When a device file is accessed, the kernel uses the major number of the file to determine which driver should be used to handle the access. This means that the kernel doesn't really need to use or even know about the minor number. The driver itself is the only thing that cares about the minor number. It uses the minor number to distinguish between different pieces of hardware.

By the way, when I say `hardware', I mean something a bit more abstract than a PCI card that you can hold in your hand. Look at these two device files:

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

By now you can look at these two device files and know instantly that they are block devices and are handled by same driver (block major `2`). You might even be aware that these both represent your floppy drive, even if you only have one floppy drive. Why two files? One represents the floppy drive with `1.44` MB of storage. The other is the *same* floppy drive with `1.68` MB of storage, and corresponds to what some people call a `superformatted' disk. One that holds more data than a standard formatted floppy. So here's a case where two

device files with different minor number actually represent the same piece of physical hardware. So just be aware that the word `hardware' in our discussion can mean something very abstract.

Chapter 4. Character Device Files

4.1. Character Device Drivers

4.1.1. The file_operations Structure

The file_operations structure is defined in `linux/fs.h`, and holds pointers to functions defined by the driver that perform various operations on the device. Each field of the structure corresponds to the address of some function defined by the driver to handle a requested operation.

For example, every character driver needs to define a function that reads from the device. The file_operations structure holds the address of the module's function that performs that operation. Here is what the definition looks like for kernel `2.6.5`:

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

Some operations are not implemented by a driver. For example, a driver that handles a video card won't need to read from a directory structure. The corresponding entries in the file_operations structure should be set to `NULL`.

There is a gcc extension that makes assigning to this structure more convenient. You'll see it in modern drivers, and may catch you by surprise. This is what the new way of assigning to the structure looks like:

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

However, there's also a C99 way of assigning to elements of a structure, and this is definitely preferred over using the GNU extension. The version of gcc I'm currently using, `2.95`, supports the new C99 syntax. You should use this syntax in case someone wants to port your driver. It will help with compatibility:

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

The meaning is clear, and you should be aware that any member of the structure which you don't explicitly assign will be initialized to `NULL` by gcc.

A pointer to a struct file_operations is commonly named `fops`.

4.1.2. The file structure

Each device is represented in the kernel by a file structure, which is defined in `linux/fs.h`. Be aware that a file is a kernel level structure and never appears in a user space program. It's not the same thing as a FILE, which is defined by glibc and would never appear in a kernel space function. Also, its name is a bit misleading; it represents an abstract open `file', not a file on a disk, which is represented by a structure named inode.

A pointer to a `struct file` is commonly named `filp`. You'll also see it refered to as `struct file file`. Resist the temptation.

Go ahead and look at the definition of `file`. Most of the entries you see, like `struct dentry` aren't used by device drivers, and you can ignore them. This is because drivers don't fill `file` directly; they only use structures contained in `file` which are created elsewhere.

4.1.3. Registering A Device

As discussed earlier, char devices are accessed through device files, usually located in `/dev`[6]. The major number tells you which driver handles which device file. The minor number is used only by the driver itself to differentiate which device it's operating on, just in case the driver handles more than one device.

Adding a driver to your system means registering it with the kernel. This is synonymous with assigning it a major number during the module's initialization. You do this by using the `register_chrdev` function, defined by `linux/fs.h`.

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

where `unsigned int major` is the major number you want to request, `const char *name` is the name of the device as it'll appear in `/proc/devices` and `struct file_operations *fops` is a pointer to the `file_operations` table for your driver. A negative return value means the registertration failed. Note that we didn't pass the minor number to `register_chrdev`. That's because the kernel doesn't care about the minor number; only our driver uses it.

Now the question is, how do you get a major number without hijacking one that's already in use? The easiest way would be to look through `Documentation/devices.txt` and pick an unused one. That's a bad way of doing things because you'll never be sure if the number you picked will be assigned later. The answer is that you can ask the kernel to assign you a dynamic major number.

If you pass a major number of 0 to `register_chrdev`, the return value will be the dynamically allocated major number. The downside is that you can't make a device file in advance, since you don't know what the major number will be. There are a couple of ways to do this. First, the driver itself can print the newly assigned number and we can make the device file by hand. Second, the newly registered device will have an entry in `/proc/devices`, and we can either make the device file by hand or write a shell script to read the file in and make the device file. The third method is we can have our driver make the the device file using the `mknod` system call after a successful registration and rm during the call to `cleanup_module`.

4.1.4. Unregistering A Device

We can't allow the kernel module to be rmmod'ed whenever root feels like it. 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 very 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's impossible because it's a void function. However, there's a counter which keeps track of how many processes are using your module. You can see what it's value is by looking at the 3rd field of `/proc/modules`. If this number isn't zero, `rmmod` will fail. Note that you don't have to check the counter from within `cleanup_module` because the check will be performed for you by the system call `sys_delete_module`, defined in `linux/module.c`. You shouldn't use this counter directly, but there are functions defined in `linux/modules.h` which let you increase, decrease and display this counter:

- `try_module_get(THIS_MODULE)`: Increment the use count.
- `try_module_put(THIS_MODULE)`: Decrement the use count.

It's important to keep the counter accurate; if you ever do lose track of the correct usage count, you'll never be able to unload the module; it's now reboot time, boys and girls. This is bound to happen to you sooner or later during a module's development.

4.1.5. chardev.c

The next code sample creates a char driver named `chardev`. You can `cat` its device file (or `open` the file with a program) and the driver will put the number of times the device file has been read from into the file. We don't support writing to the file (like **echo "hi" > /dev/hello**), but catch these attempts and tell the user that the operation isn't supported. Don't worry if you don't see what we do with the data we read into the

buffer; we don't do much with it. We simply read in the data and print a message acknowledging that we received it.

Example 4–1. chardev.c

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

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

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

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

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

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

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

/*
 * Functions
 */

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

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

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

        return 0;
}

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

/*
 * Methods
 */

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

        return SUCCESS;
}

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

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

        return 0;
}

/*
 * Called when a process, which already opened the dev file, attempts to
 * read from it.
 */
static ssize_t device_read(struct file *filp,   /* see include/linux/fs.h   */
```

```
                           char *buffer,        /* buffer to fill with data */
                           size_t length,       /* length of the buffer     */
                           loff_t * offset)
{
        /*
         * Number of bytes actually written to the buffer
         */
        int bytes_read = 0;

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

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

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

                length--;
                bytes_read++;
        }

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

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

4.1.6. Writing Modules for Multiple Kernel Versions

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).

There are differences between different kernel versions, and if you want to support multiple kernel versions, you'll find yourself having to code conditional compilation directives. The way to do this to compare the macro `LINUX_VERSION_CODE` to the macro `KERNEL_VERSION`. In version `a.b.c` of the kernel, the value of this macro would be $2^{16}a+2^{8}b+c$.

While previous versions of this guide showed how you can write backward compatible code with such constructs in great detail, we decided to break with this tradition for the better. People interested in doing such might now use a LKMPG with a version matching to their kernel. We decided to version the LKMPG like the kernel, at least as far as major and minor number are concerned. We use the patchlevel for our own versioning so use LKMPG version 2.4.x for kernels 2.4.x, use LKMPG version 2.6.x for kernels 2.6.x and so on. Also make sure that you always use current, up to date versions of both, kernel and guide.

Chapter 5. The /proc File System

5.1. 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`[7] 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 `try_module_get` and `try_module_put` 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 `/proc` files which will allow us to protect against this problem as well.

Example 5−1. procfs.c

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

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

struct proc_dir_entry *Our_Proc_File;

/* Put data into the proc fs file.
 *
 * Arguments
 * =========
 * 1. The buffer where the data is to be inserted, if
 *    you decide to use it.
 * 2. A pointer to a pointer to characters. This is
 *    useful if you don't want to use the buffer
 *    allocated by the kernel.
 * 3. The current position in the file
 * 4. The size of the buffer in the first argument.
 * 5. Write a "1" here to indicate EOF.
 * 6. A pointer to data (useful in case one common
 *    read for multiple /proc/... entries)
 *
```

```
 * Usage and Return Value
 * ======================
 * A return value of zero means you have no further
 * information at this time (end of file). A negative
 * return value is an error condition.
 *
 * For More Information
 * ====================
 * The way I discovered what to do with this function
 * wasn't by reading documentation, but by reading the
 * code which used it. I just looked to see what uses
 * the get_info field of proc_dir_entry struct (I used a
 * combination of find and grep, if you're interested),
 * and I saw that  it is used in <kernel source
 * directory>/fs/proc/array.c.
 *
 * If something is unknown about the kernel, this is
 * usually the way to go. In Linux we have the great
 * advantage of having the kernel source code for
 * free - use it.
 */
ssize_t
procfile_read(char *buffer,
              char **buffer_location,
              off_t offset, int buffer_length, int *eof, void *data)
{
        printk(KERN_INFO "inside /proc/test : procfile_read\n");

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

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

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

        /*
         * Return the length
```

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

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

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

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

        return rv;
}

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

Chapter 6. Using /proc For Input

6.1. 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 `struct proc_dir_entry` 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[8], 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 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 user (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.

Example 6−1. procfs.c

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

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

#define PROC_ENTRY_FILENAME "rw_test"

static ssize_t module_output(struct file *filp, /* see include/linux/fs.h   */
                             char *buffer,      /* buffer to fill with data */
                             size_t length,     /* length of the buffer     */
                             loff_t * offset)
{
        static int finished = 0;
        int i;
        char message[MESSAGE_LENGTH + 30];

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

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

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

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

static ssize_t
module_input(struct file *filp, const char *buff, size_t len, loff_t * off)
```

```
{
        int i;
        /*
         * Put the input into Message, where module_output
         * will later be able to use it
         */
        for (i = 0; i < MESSAGE_LENGTH - 1 && i < len; i++)
                get_user(Message[i], buff + i);

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

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

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

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

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

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

```
}

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

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

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

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

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

        return rv;
}

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

Still hungry for procfs examples? Well, first of all keep in mind, there are rumors around, claiming that procfs is on it's way out, consider using sysfs instead. Second, if you really can't get enough, there's a highly recommendable bonus level for procfs below `linux/Documentation/DocBook/` . Use **make help** in your toplevel kernel directory for instructions about how to convert it into your favourite format. Example: **make htmldocs** . Consider using this mechanism, in case you want to document something kernel related yourself.

Chapter 7. Talking To Device Files

7.1. 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 Input Output ConTroL). 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), [9] 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 long so you can use a cast to use it to pass anything. [10]

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 included 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`.

Example 7−1. chardev.c

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

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

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

/*
 * Is the device open right now? Used to prevent
```

```
 * concurent access into the same device
 */
static int Device_Open = 0;

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

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

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

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

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

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

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

        module_put(THIS_MODULE);
        return SUCCESS;
}

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

```
                           size_t length,       /* length of the buffer     */
                           loff_t * offset)
{
        /*
         * Number of bytes actually written to the buffer
         */
        int bytes_read = 0;

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

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

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

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

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

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

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

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

        for (i = 0; i < length && i < BUF_LEN; i++)
```

```
                get_user(Message[i], buffer + i);

        Message_Ptr = Message;

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

/*
 * This function is called whenever a process tries to do an ioctl on our
 * device file. We get two extra parameters (additional to the inode and file
 * structures, which all device functions get): the number of the ioctl called
 * and the parameter given to the ioctl function.
 *
 * If the ioctl is write or read/write (meaning output is returned to the
 * calling process), the ioctl call returns the output of this function.
 *
 */
int device_ioctl(struct inode *inode,   /* see include/linux/fs.h */
                 struct file *file,     /* ditto */
                 unsigned int ioctl_num,        /* number and param for ioctl */
                 unsigned long ioctl_param)
{
        int i;
        char *temp;
        char ch;

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

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

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

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

                /*
                 * Put a zero at the end of the buffer, so it will be
                 * properly terminated
                 */
```

```
                put_user('\0', (char *)ioctl_param + i);
                break;

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

        return SUCCESS;
}

/* Module Declarations */

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

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

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

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

```
        return 0;
}

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

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

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

Example 7–2. chardev.h

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

#ifndef CHARDEV_H
#define CHARDEV_H

#include <linux/ioctl.h>

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

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

```
 * the process to the kernel.
 */

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

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

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

#endif
```

Example 7–3. ioctl.c

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

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

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

/*
 * Functions for the ioctl calls
 */

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

        ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);
```

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

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

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

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

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

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

        printf("get_nth_byte message:");

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

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

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

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

        file_desc = open(DEVICE_FILE_NAME, 0);
        if (file_desc < 0) {
```

```
                printf("Can't open device file: %s\n", DEVICE_FILE_NAME);
                exit(-1);
        }

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

        close(file_desc);
}
```

Chapter 8. System Calls

8.1. 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 <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.

Note that all the related problems make syscall stealing unfeasiable for production use. In order to keep people from doing potential harmful things sys_call_table is no longer exported. This means, if you want to do something more than a mere dry run of this example, you will have to patch your current kernel in order to have sys_call_table exported. In the example directory you will find a README and the patch. As you can imagine, such modifications are not to be taken lightly. Do not try this on valueable systems (ie systems that you do not own – or cannot restore easily). It should go O.K., but kernel people, choose to not support hacks like this in 2.6. and I'm sure they've had their reasons for that. For more details, see the README. If in doubt, saying N here and skipping this example is the safe choice.

Example 8−1. syscall.c

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

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

/*
 * The necessary header files
 */

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

/*
 * For the current (process) structure, we need
```

```
 * this to know who the current user is.
 */
#include <linux/sched.h>
#include <asm/uaccess.h>

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

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

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

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

        /*
         * Check if this is the user we're spying on
         */
        if (uid == current->uid) {
```

```
                /*
                 * Report the file, if relevant
                 */
                printk("Opened file by %d: ", uid);
                do {
                        get_user(ch, filename + i);
                        i++;
                        printk("%c", ch);
                } while (ch != 0);
                printk("\n");
        }

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

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

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

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

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

        return 0;
}

/*
 * Cleanup - unregister the appropriate file from /proc
 */
void cleanup_module()
{
        /*
         * Return the system call back to normal
         */
```

```
        if (sys_call_table[__NR_open] != our_sys_open) {
                printk("Somebody else also played with the ");
                printk("open system call\n");
                printk("The system may be left in ");
                printk("an unstable state.\n");
        }

        sys_call_table[__NR_open] = original_call;
}
```

Chapter 9. Blocking Processes

9.1. Blocking Processes

9.1.1. Enter Sandman

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 `wait_event_interruptible`[11]. 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`[12]. It can then proceed to set a global variable to 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. [13] 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`*.

Example 9–1. sleep.c

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

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

```
#include <linux/sched.h>        /* For putting processes to sleep and
                                   waking them up */
#include <asm/uaccess.h>        /* for get_user and put_user */

/*
 * The module's file functions
 */

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

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

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

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

        /*
         * If you don't understand this by now, you're hopeless as a kernel
         * programmer.
         */
        sprintf(message, "Last input:%s\n", Message);
        for (i = 0; i < len && message[i]; i++)
                put_user(message[i], buf + i);

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

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

```
{                               /* offset to file - ignore */
        int i;

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

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

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

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

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

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

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

                /*
                 * This function puts the current process, including any system
                 * calls, such as us, to sleep.  Execution will be resumed right
                 * after the function call, either because somebody called
                 * wake_up(&WaitQ) (only module_close does that, when the file
```

```
                 * is closed) or when a signal, such as Ctrl-C, is sent
                 * to the process
                 */
                wait_event_interruptible(WaitQ, !Already_Open);

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

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

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

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

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

/*
 * Called when the /proc file is closed
 */
int module_close(struct inode *inode, struct file *file)
{
        /*
```

```
         * Set Already_Open to zero, so one of the processes in the WaitQ will
         * be able to set Already_Open back to one and to open the file. All
         * the other processes will be called when Already_Open is back to one,
         * so they'll go back to sleep.
         */
        Already_Open = 0;

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

        module_put(THIS_MODULE);

        return 0;               /* success */
}

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

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

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

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

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

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

/*
 * Module initialization and cleanup
 */

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

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

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

        return rv;
}

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

Chapter 10. Replacing Printks

10.1. Replacing `printk`

In Section 1.2.1.2, I said that X and kernel module programming don't mix. That's true for developing kernel modules, but in actual use, you want to be able to send messages to whichever tty[14] the command to load the module came from.

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.

Example 10–1. print_string.c

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

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

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

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

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

                /*
                 * my_tty->driver is a struct which holds the tty's functions,
```

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

                /*
                 * ttys were originally hardware devices, which (usually)
                 * strictly followed the ASCII standard.  In ASCII, to move to
                 * a new line you need two characters, a carriage return and a
                 * line feed.  On Unix, the ASCII line feed is used for both
                 * purposes - so we can't just use \n, because it wouldn't have
                 * a carriage return and the next line will start at the
                 * column right after the line feed.
                 *
                 * This is why text files are different between Unix and
                 * MS Windows.  In CP/M and derivatives, like MS-DOS and
                 * MS Windows, the ASCII standard was strictly adhered to,
                 * and therefore a newline requirs both a LF and a CR.
                 */
                ((my_tty->driver)->write) (my_tty, 0, "\015\012", 2);
        }
}

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

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

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

10.2. Flashing keyboard LEDs

In certain conditions, you may desire for your module a simpler and more direct way to communicate to the external world that he's running. Flashing keyboard LEDs can be such a solution: It is an immediate way to attract attention or to display a status condition. Keyboard LEDs are present on every hardware, they are always visible, they do not need any setup, and their use is rather simple and non−intrusive, if compared to writing to a tty or a file.

The following source code illustrates a minimal kernel module which, when loaded, starts blinking the keyboard LEDs until it is unloaded.

Example 10–2. kbleds.c

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

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

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

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

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

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

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

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

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

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

static int __init kbleds_init(void)
```

```
{
        int i;

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

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

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

        return 0;
}

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

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

If none of the examples in this chapter fit your debugging needs there might yet be some other tricks to try. Ever wondered what CONFIG_LL_DEBUG in **make menuconfig** is good for? If you activate that you get low level access to the serial port. While this might not sound very powerful by itself, you can patch `kernel/printk.c` or any other essential syscall to use printascii, thus makeing it possible to trace virtually everything what your code does over a serial line. If your architecture does not support this and is equipped with a serial port, this might be one of the first things that should be implemented. Logging over a netconsole might also be worth a try.

While you have seen lots of stuff that can be used to aid debugging here, there are some things to be aware of. Debugging is almost always intrusive. Adding debug code can change the situation enough to make the bug seem to dissappear. Thus you should try to keep debug code to a minimum and make shure it does not show up in production code.

Chapter 11. Scheduling Tasks

11.1. 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 tq_struct structure, 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`[15] 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.

Example 11−1. sched.c

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

/*
 * The necessary header files
 */

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

```
#include <linux/workqueue.h>    /* We scheduale tasks here */
#include <linux/sched.h>        /* We need to put ourselves to sleep
                                   and wake up later */
#include <linux/init.h>         /* For __init and __exit */
#include <linux/interrupt.h>    /* For irqreturn_t */

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

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

static void intrpt_routine(void *);

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

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

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

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

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

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

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

```
        static int count = 1;

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

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

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

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

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

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

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

        return rv;
}

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

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

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

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

}

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

Chapter 12. Interrupt Handlers

12.1. Interrupt Handlers

12.1.1. 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 IRQ's (*I*nterrupt *Req*uests)[16]. There are two types of IRQ's, 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 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 15 of them, plus 1 which is used to cascade the interrupt controllers, 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`[17]. 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.

12.1.2. Keyboards on the Intel Architecture

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.

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 thinks 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).

Example 12–1. intrpt.c

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

/*
 * The necessary header files
 */

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

#define MY_WORK_QUEUE_NAME "WQsched.c"

static struct workqueue_struct *my_workqueue;

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

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

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

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

        queue_work(my_workqueue, &task);

        return IRQ_HANDLED;
}

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

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

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

/*
```

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

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

Chapter 13. Symmetric Multi Processing

13.1. Symmetrical Multi−Processing

One of the easiest and cheapest ways to improve hardware performance is to put more than one CPU on the board. This can be done either making the different CPU's 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 CPU's 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[18]. The kernel, on the other hand, could be called by different processes running on different CPU's.

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[19], but inefficient.

In version 2.2.x, several CPU's can be in the kernel at the same time. This is something module writers need to be aware of.

Chapter 14. Common Pitfalls

14.1. 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 happens, please report the problem to me for a full refund of the amount I was paid for your copy of the book.

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/kallsyms`.

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.

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: 2.0 To 2.2

A.1. Changes between 2.0 and 2.2

A.1.1. 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 located on Richard Gooch's site .

asm/uaccess.h
: If you need `put_user` or `get_user` you have to **`#include`** it.

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.

file_operations
: This structure now has a flush function between the `open` and `close` functions.

`close` in file_operations
: In version 2.2, the `close` function returns an integer, so it's allowed to fail.

`read`,`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.

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.

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

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`.

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.

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 To Go From Here

B.1. 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 protocol stacks (as if there's a need for that −− you'd have to dig underground 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 Juan−Mariano de Goyeneche's list of kernel resources . Also, as Linus said, the best way 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.