Linux Device Drivers, 3rd Edition

By Jonathan Corbet, Greg Kroah-Hartman, Alessandro Rubini

Publisher: O'Reilly
Pub Date: February 2005
ISBN: 0-596-00590-3
Pages: 636

Over the years, this bestselling guide has helped countless programmers learn how to support computer peripherals under the Linux operating system, and how to develop new hardware under Linux. Now, with this third edition, it's even more helpful, covering all the significant changes to Version 2.6 of the Linux kernel. Includes full-featured examples that programmers can compile and run without special hardware.

Linux Device Drivers, 3rd Edition
By Jonathan Corbet, Greg Kroah-Hartman, Alessandro Rubini

Publisher: O'Reilly
Pub Date: February 2005
ISBN: 0-596-00590-3
Pages: 636

- Table of Contents
- Index
- Reviews
- Examples
- Reader Reviews
- Errata
- Academic

Preface

This is, on the surface, a book about writing device drivers for the Linux system. That is a worthy goal, of course; the flow of new hardware products is not likely to slow down anytime soon, and somebody is going to have to make all those new gadgets work with Linux. But this book is also about how the Linux kernel works and how to adapt its workings to your needs or interests. Linux is an open system; with this book, we hope, it is more open and accessible to a larger community of developers.

This is the third edition of Linux Device Drivers. The kernel has changed greatly since this book was first published, and we have tried to evolve the text to match. This edition covers the 2.6.10 kernel as completely as we are able. We have, this time around, elected to omit the discussion of backward compatibility with previous kernel versions. The changes from 2.4 are simply too large, and the 2.4 interface remains well documented in the (freely available) second edition.

This edition contains quite a bit of new material relevant to the 2.6 kernel. The discussion of locking and concurrency has been expanded and moved into its own chapter. The Linux device model, which is new in 2.6, is covered in detail. There are new chapters on the USB bus and the serial driver subsystem; the chapter on PCI has also been enhanced. While the organization of the rest of the book resembles that of the earlier editions, every chapter has been thoroughly updated.

We hope you enjoy reading this book as much as we have enjoyed writing it.

Jon's Introduction

The publication of this edition coincides with my twelth year of working with Linux and, shockingly, my twenty-fifth year in the computing field. Computing seemed like a fast-moving field back in 1980, but things have sped up a lot since then. Keeping Linux Device Drivers up to date is increasingly a challenge; the Linux kernel hackers continue to improve their code, and they have little patience for documentation that fails to keep up.

Linux continues to succeed in the market and, more importantly, in the hearts and minds of developers worldwide. The success of Linux is clearly a testament to its technical quality and to the numerous benefits of free software in general. But the true key to its success, in my opinion, lies in the fact that it has brought the fun back to computing. With Linux, anybody can get their hands into the system and play in a sandbox where contributions from any direction are welcome, but where technical excellence is valued above all else. Linux not only provides us with a top-quality operating system; it gives us the opportunity to be part of its future development and to have fun while we're at it.

In my 25 years in the field, I have had many interesting opportunities, from programming the first Cray computers (in Fortran, on punch cards) to seeing the minicomputer and Unix workstation waves, through to the current, microprocessor-dominated era. Never, though, have I seen the field more full of life, opportunity, and fun. Never have we had such control over our own tools and their evolution. Linux, and free software in general, is clearly the driving force behind those changes.

My hope is that this edition helps to bring that fun and opportunity to a new set of Linux developers. Whether your interests are in the kernel or in user space, I hope you find this book to be a useful and interesting guide to just how the kernel works with the hardware. I hope it helps and inspires you to fire up your editor and to make our shared, free operating system even better. Linux has come a long way, but it is also just beginning; it will be more than interesting to watch—and participate in—what happens from here.

Alessandro's Introduction

I've always enjoyed computers because they can talk to external hardware. So, after soldering my devices for the Apple II and the ZX Spectrum, backed with the Unix and free software expertise the university gave me, I could escape the DOS trap by installing GNU/Linux on a fresh new 386 and by turning on the soldering iron once again.

Back then, the community was a small one, and there wasn't much documentation about writing drivers around, so I started writing for Linux Journal. That's how things started: when I later discovered I didn't like writing papers, I left the univeristy and found myself with an O'Reilly contract in my hands.

That was in 1996. Ages ago.

The computing world is different now: free software looks like a viable solution, both technically and politically, but there's a lot of work to do in both realms. I hope this book furthers two aims: spreading technical knowledge and raising awareness about the need to spread knowledge. That's why, after the first edition proved interesting to the public, the two authors of the second edition switched to a free license, supported by our editor and our publisher. I'm betting this is the right approach to information, and it's great to team up with other people sharing this vision.

I'm excited by what I witness in the embedded arena, and I hope this text helps by doing more; but ideas are moving fast these days, and it's already time to plan for the fourth edition, and look for a fourth author to help.

Greg's Introduction

It seems like a long time ago that I picked up the first edition of this Linux Device Drivers book in order to figure out how to write a real Linux driver. That first edition was a great guide to helping me understand the internals of this operating system that I had already been using for a number of years but whose kernel had never taken the time to look into. With the knowledge gained from that book, and by reading other programmers' code already present in the kernel, my first horribly buggy, broken, and very SMP-unsafe driver was accepted by the kernel community into the main kernel tree. Despite receiving my first bug report five minutes later, I was hooked on wanting to do as much as I could to make this operating system the best it could possibly be.

I am honored that I've had the ability to contribute to this book. I hope that it enables others to learn the details about the kernel, discover that driver development is not a scary or forbidding place, and possibly encourage others to join in and help in the collective effort of making this operating system work on every computing platform with every type of device available. The development procedure is fun, the community is rewarding, and everyone benefits from the effort involved.

Now it's back to making this edition obsolete by fixing current bugs, changing APIs to work better and be simpler to understand for everyone, and adding new features. Come along; we can always use the help.

Audience for This Book

This book should be an interesting source of information both for people who want to experiment with their computer and for technical programmers who face the need to deal with the inner levels of a Linux box. Note that "a Linux box" is a wider concept than "a PC running Linux," as many platforms are supported by our operating system, and kernel programming is by no means bound to a specific platform. We hope this book is useful as a starting point for people who want to become kernel hackers but don't know where to start.

On the technical side, this text should offer a hands-on approach to understanding the kernel internals and some of the design choices made by the Linux developers. Although the main, official target of the book is teaching how to write device drivers, the material should give an interesting overview of the kernel implementation as well.

Although real hackers can find all the necessary information in the official kernel sources, usually a written text can be helpful in developing programming skills. The text you are approaching is the result of hours of patient grepping through the kernel sources, and we hope the final result is worth the effort it took.

The Linux enthusiast should find in this book enough food for her mind to start playing with the code base and should be able to join the group of developers that is continuously working on new capabilities and performance enhancements. This book does not cover the Linux kernel in its entirety, of course, but Linux device driver authors need to know how to work with many of the kernel's subsystems. Therefore, it makes a good introduction to kernel programming in general. Linux is still a work in progress, and there's always a place for new programmers to jump into the game.

If, on the other hand, you are just trying to write a device driver for your own device, and you don't want to muck with the kernel internals, the text should be modularized enough to fit your needs as well. If you don't want to go deep into the details, you can just skip the most technical sections, and stick to the standard API used by device drivers to seamlessly integrate with the rest of the kernel.

Organization of the Material

The book introduces its topics in ascending order of complexity and is divided into two parts. The first part (Chapters 1-11) begins with the proper setup of kernel modules and goes on to describe the various aspects of programming that you'll need in order to write a full-featured driver for a char-oriented device. Every chapter covers a distinct problem and includes a quick summary at the end, which can be used as a reference during actual development.

Throughout the first part of the book, the organization of the material moves roughly from the software-oriented concepts to the hardware-related ones. This organization is meant to allow you to test the software on your own computer as far as possible without the need to plug external hardware into the machine. Every chapter includes source code and points to sample drivers that you can run on any Linux computer. In Chapter 9 and Chapter 10, however, we ask you to connect an inch of wire to the parallel port in order to test out hardware handling, but this requirement should be manageable by everyone.

The second half of the book (Chapters 12-18) describes block drivers and network interfaces and goes deeper into more advanced topics, such as working with the virtual memory subsystem and with the PCI and USB buses. Many driver authors do not need all of this material, but we encourage you to go on reading anyway. Much of the material found there is interesting as a view into how the Linux kernel works, even if you do not need it for a specific project.

Background Information

In order to be able to use this book, you need to be confident with C programming. Some Unix expertise is needed as well, as we often refer to Unix semantics about system calls, commands, and pipelines.

At the hardware level, no previous expertise is required to understand the material in this book, as long as the general concepts are clear in advance. The text isn't based on specific PC hardware, and we provide all the needed information when we do refer to specific hardware.

Several free software tools are needed to build the kernel, and you often need specific versions of these tools. Those that are too old can lack needed features, while those that are too new can occasionally generate broken kernels. Usually, the tools provided with any current distribution work just fine. Tool version requirements vary from one kernel to the next; consult *Documentation/Changes* in the source tree of the kernel you are using for exact requirements.

Online Version and License

Conventions Used in This Book

The following is a list of the typographical conventions used in this book:

Italic

Used for file and directory names, program and command names, command-line options, URLs, and new terms

Constant Width

Used in examples to show the contents of files or the output from commands, and in the text to indicate words that appear in C code or other literal strings

Constant Width Italic

Used to indicate text within commands that the user replaces with an actual value

`Constant Width Bold`

Used in examples to show commands or other text that should be typed literally by the user

Pay special attention to notes set apart from the text with the following icons:

This is a tip. It contains useful supplementary information about the topic at hand.

This is a warning. It helps you solve and avoid annoying problems.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. The code samples are covered by a dual BSD/GPL license.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "Linux Device Drivers, Third Edition, by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. Copyright 2005 O'Reilly Media, Inc., 0-596-00590-3."

We'd Like to Hear from You

Please address comments and questions concerning this book to the publisher:
O'Reilly Media, Inc.1005 Gravenstein Highway NorthSebastopol, CA 95472(800) 998-9938 (in the United States or Canada)(707) 829-0515 (international or local)(707) 829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:
http://www.oreilly.com/catalog/linuxdrive3

To comment or ask technical questions about this book, send email to:
bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:
http://www.oreilly.com

Safari Enabled

When you see a Safari® Enabled icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at http://safari.oreilly.com.

Acknowledgments

This book, of course, was not written in a vacuum; we would like to thank the many people who have helped to make it possible.

Thanks to our editor, Andy Oram; this book is a vastly better product as a result of his efforts. And obviously we owe a lot to the smart people who have laid the philosophical and practical foundations of the current free software renaissance.

The first edition was technically reviewed by Alan Cox, Greg Hankins, Hans Lermen, Heiko Eissfeldt, and Miguel de Icaza (in alphabetic order by first name). The technical reviewers for the second edition were Allan B. Cruse, Christian Morgner, Jake Edge, Jeff Garzik, Jens Axboe, Jerry Cooperstein, Jerome Peter Lynch, Michael Kerrisk, Paul Kinzelman, and Raph Levien. Reviewers for the third edition were Allan B. Cruse, Christian Morgner, James Bottomley, Jerry Cooperstein, Patrick Mochel, Paul Kinzelman, and Robert Love. Together, these people have put a vast amount of effort into finding problems and pointing out possible improvements to our writing.

Last but certainly not least, we thank the Linux developers for their relentless work. This includes both the kernel programmers and the user-space people, who often get forgotten. In this book, we chose never to call them by name in order to avoid being unfair to someone we might forget. We sometimes made an exception to this rule and called Linus by name; we hope he doesn't mind.

Jon

I must begin by thanking my wife Laura and my children Michele and Giulia for filling my life with joy and patiently putting up with my distraction while working on this edition. The subscribers of LWN.net have, through their generosity, enabled much of this work to happen. The Linux kernel developers have done me a great service by letting me be a part of their community, answering my questions, and setting me straight when I got confused. Thanks are due to readers of the second edition of this book whose comments, offered at Linux gatherings over much of the world, have been gratifying and inspiring. And I would especially like to thank Alessandro Rubini for starting this whole exercise with the first edition (and staying with it through the current edition); and Greg Kroah-Hartman, who has brought his considerable skills to bear on several chapters, with great results.

Alessandro

I would like to thank the people that made this work possible. First of all, the incredible patience of Federica, who went as far as letting me review the first edition during our honeymoon, with a laptop in the tent. I want to thank Giorgio and Giulia, who have been involved in later editions of the book and happily accepted to be sons of "a gnu" who often works late in the night. I owe a lot to all the free-software authors who actually taught me how to program by making their work available for anyone to study. But for this edition, I'm mostly grateful to Jon and Greg, who have been great mates in this work; it couldn't have existed without each and both of them, as the code base is bigger and tougher, while my time is a scarcer resource, always contended for by clients, free software issues, and expired deadlines. Jon has been a great leader for this edition; both have been very productive and technically invaluable in supplementing my small-scale and embedded view toward programming with their expertise about SMP and number crunchers.

Greg

I would like to thank my wife Shannon and my children Madeline and Griffin for their understanding and patience while I took the time to work on this book. If it were not for their support of my original Linux development efforts, I would not be able to do this book at all. Thanks also to Alessandro and Jon for offering to let me work on this book; I am honored that they let me participate in it. Much gratitude is given to all of the Linux kernel programmers, who were unselfish enough to write code in the public view, so that I and others could learn so much from just reading it. Also, for everyone who has ever sent me bug reports, critiqued my code, and flamed me for doing stupid things, you have all taught me so much about how to be a better programmer and, throughout it all, made me feel very welcome to be part of this community. Thank you.

Chapter 1. An Introduction to Device Drivers

One of the many advantages of free operating systems, as typified by Linux, is that their internals are open for all to view. The operating system, once a dark and mysterious area whose code was restricted to a small number of programmers, can now be readily examined, understood, and modified by anybody with the requisite skills. Linux has helped to democratize operating systems. The Linux kernel remains a large and complex body of code, however, and would-be kernel hackers need an entry point where they can approach the code without being overwhelmed by complexity. Often, device drivers provide that gateway.

Device drivers take on a special role in the Linux kernel. They are distinct "black boxes" that make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works. User activities are performed by means of a set of standardized calls that are independent of the specific driver; mapping those calls to device-specific operations that act on real hardware is then the role of the device driver. This programming interface is such that drivers can be built separately from the rest of the kernel and "plugged in" at runtime when needed. This modularity makes Linux drivers easy to write, to the point that there are now hundreds of them available.

There are a number of reasons to be interested in the writing of Linux device drivers. The rate at which new hardware becomes available (and obsolete!) alone guarantees that driver writers will be busy for the foreseeable future. Individuals may need to know about drivers in order to gain access to a particular device that is of interest to them. Hardware vendors, by making a Linux driver available for their products, can add the large and growing Linux user base to their potential markets. And the open source nature of the Linux system means that if the driver writer wishes, the source to a driver can be quickly disseminated to millions of users.

This book teaches you how to write your own drivers and how to hack around in related parts of the kernel. We have taken a device-independent approach; the programming techniques and interfaces are presented, whenever possible, without being tied to any specific device. Each driver is different; as a driver writer, you need to understand your specific device well. But most of the principles and basic techniques are the same for all drivers. This book cannot teach you about your device, but it gives you a handle on the background you need to make your device work.

As you learn to write drivers, you find out a lot about the Linux kernel in general; this may help you understand how your machine works and why things aren't always as fast as you expect or don't do quite what you want. We introduce new ideas gradually, starting off with very simple drivers and building on them; every new concept is accompanied by sample code that doesn't need special hardware to be tested.

This chapter doesn't actually get into writing code. However, we introduce some background concepts about the Linux kernel that you'll be glad you know later, when we do launch into programming.

1.1. The Role of the Device Driver

As a programmer, you are able to make your own choices about your driver, and choose an acceptable trade-off between the programming time required and the flexibility of the result. Though it may appear strange to say that a driver is "flexible," we like this word because it emphasizes that the role of a device driver is providing mechanism, not policy.

The distinction between mechanism and policy is one of the best ideas behind the Unix design. Most programming problems can indeed be split into two parts: "what capabilities are to be provided" (the mechanism) and "how those capabilities can be used" (the policy). If the two issues are addressed by different parts of the program, or even by different programs altogether, the software package is much easier to develop and to adapt to particular needs.

For example, Unix management of the graphic display is split between the X server, which knows the hardware and offers a unified interface to user programs, and the window and session managers, which implement a particular policy without knowing anything about the hardware. People can use the same window manager on different hardware, and different users can run different configurations on the same workstation. Even completely different desktop environments, such as KDE and GNOME, can coexist on the same system. Another example is the layered structure of TCP/IP networking: the operating system offers the socket abstraction, which implements no policy regarding the data to be transferred, while different servers are in charge of the services (and their associated policies). Moreover, a server like ftpd provides the file transfer mechanism, while users can use whatever client they prefer; both command-line and graphic clients exist, and anyone can write a new user interface to transfer files.

Where drivers are concerned, the same separation of mechanism and policy applies. The floppy driver is policy free—its role is only to show the diskette as a continuous array of data blocks. Higher levels of the system provide policies, such as who may access the floppy drive, whether the drive is accessed directly or via a filesystem, and whether users may mount filesystems on the drive. Since different environments usually need to use hardware in different ways, it's important to be as policy free as possible.

When writing drivers, a programmer should pay particular attention to this fundamental concept: write kernel code to access the hardware, but don't force particular policies on the user, since different users have different needs. The driver should deal with making the hardware available, leaving all the issues about how to use the hardware to the applications. A driver, then, is flexible if it offers access to the hardware capabilities without adding constraints. Sometimes, however, some policy decisions must be made. For example, a digital I/O driver may only offer byte-wide access to the hardware in order to avoid the extra code needed to handle individual bits.

You can also look at your driver from a different perspective: it is a software layer that lies between the applications and the actual device. This privileged role of the driver allows the driver programmer to choose exactly how the device should appear: different drivers can offer different capabilities, even for the same device. The actual driver design should be a balance between many different considerations. For instance, a single device may be used concurrently by different programs, and the driver programmer has complete freedom to determine how to handle concurrency. You could implement memory mapping on the device independently of its hardware capabilities, or you could provide a user library to help application programmers implement new policies on top of the available primitives, and so forth. One major consideration is the trade-off between the desire to present the user with as many options as possible and the time you have to write the driver, as well as the need to keep things simple so that errors don't creep in.

Policy-free drivers have a number of typical characteristics. These include support for both synchronous and asynchronous operation, the ability to be opened multiple times, the ability to exploit the full capabilities of the hardware, and the lack of software layers to "simplify things" or provide policy-related operations. Drivers of this sort not only work better for their end users, but also turn out to be easier to write and maintain as well. Being policy-free is actually a common target for software designers.

Many device drivers, indeed, are released together with user programs to help with configuration and access to the target device. Those programs can range from simple utilities to complete graphical applications. Examples include the tunelp program, which adjusts how the parallel port printer driver operates, and the graphical cardctl utility that is part of the PCMCIA driver package. Often a client library is provided as well, which provides capabilities that do not need to be implemented as part of the driver itself.

1.2. Splitting the Kernel

In a Unix system, several concurrent *processes* attend to different tasks. Each process asks for system resources, be it computing power, memory, network connectivity, or some other resource. The kernel is the big chunk of executable code in charge of handling all such requests. Although the distinction between the different kernel tasks isn't always clearly marked, the kernel's role can be split (as shown in Figure 1-1) into the following parts:

Process management

The kernel is in charge of creating and destroying processes and handling their connection to the outside world (input and output). Communication among different processes (through signals, pipes, or interprocess communication primitives) is basic to the overall system functionality and is also handled by the kernel. In addition, the scheduler, which controls how processes share the CPU, is part of process management. More generally, the kernel's process management activity implements the abstraction of several processes on top of a single CPU or a few of them.

Memory management

The computer's memory is a major resource, and the policy used to deal with it is a critical one for system performance. The kernel builds up a virtual addressing space for any and all processes on top of the limited available resources. The different parts of the kernel interact with the memory-management subsystem through a set of function calls, ranging from the simple malloc/free pair to much more complex functionalities.

Filesystems

Unix is heavily based on the filesystem concept; almost everything in Unix can be treated as a file. The kernel builds a structured filesystem on top of unstructured hardware, and the resulting file abstraction is heavily used throughout the whole system. In addition, Linux supports multiple filesystem types, that is, different ways of organizing data on the physical medium. For example, disks may be formatted with the Linux-standard ext3 filesystem, the commonly used FAT filesystem or several others.

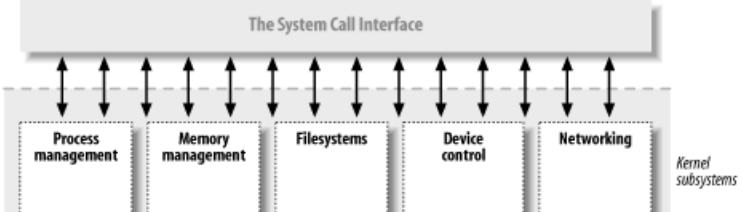
Device control

Almost every system operation eventually maps to a physical device. With the exception of the processor, memory, and a very few other entities, any and all device control operations are performed by code that is specific to the device being addressed. That code is called a *device driver*. The kernel must have embedded in it a device driver for every peripheral present on a system, from the hard drive to the keyboard and the tape drive. This aspect of the kernel's functions is our primary interest in this book.

Networking

Networking must be managed by the operating system, because most network operations are not specific to a process: incoming packets are asynchronous events. The packets must be collected, identified, and dispatched before a process takes care of them. The system is in charge of delivering data packets across program and network interfaces, and it must control the execution of programs according to their network activity. Additionally, all the routing and address resolution issues are implemented within the kernel.

Figure 1-1. A split view of the kernel



1.3. Classes of Devices and Modules

The Linux way of looking at devices distinguishes between three fundamental device types. Each module usually implements one of these types, and thus is classifiable as a char module, a block module, or a network module. This division of modules into different types, or classes, is not a rigid one; the programmer can choose to build huge modules implementing different drivers in a single chunk of code. Good programmers, nonetheless, usually create a different module for each new functionality they implement, because decomposition is a key element of scalability and extendability.

The three classes are:

Character devices

A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the open, close, read, and write system calls. The text console (*/dev/console*) and the serial ports (*/dev/ttyS0* and friends) are examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of filesystem nodes, such as /dev/tty1 and /dev/lp0. The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially. There exist, nonetheless, char devices that look like data areas, and you can move back and forth in them; for instance, this usually applies to frame grabbers, where the applications can access the whole acquired image using mmap or lseek.

Block devices

Like char devices, block devices are accessed by filesystem nodes in the */dev* directory. A block device is a device (e.g., a disk) that can host a filesystem. In most Unix systems, a block device can only handle I/O operations that transfer one or more whole blocks, which are usually 512 bytes (or a larger power of two) bytes in length. Linux, instead, allows the application to read and write a block device like a char device—it permits the transfer of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface. Like a char device, each block device is accessed through a filesystem node, and the difference between them is transparent to the user. Block drivers have a completely different interface to the kernel than char drivers.

Network interfaces

Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. Usually, an *interface* is a hardware device, but it might also be a pure software device, like the loopback interface. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Many network connections (especially those using TCP) are stream-oriented, but network devices are, usually, designed around the transmission and receipt of packets. A network driver knows nothing about individual connections; it only handles packets.

Not being a stream-oriented device, a network interface isn't easily mapped to a node in the filesystem, as /dev/tty1 is. The Unix way to provide access to interfaces is still by assigning a unique name to them (such as eth0), but that name doesn't have a corresponding entry in the filesystem. Communication between the kernel and a network device driver is completely different from that used with char and block drivers. Instead of read and write, the kernel calls functions related to packet transmission.

There are other ways of classifying driver modules that are orthogonal to the above device types. In general, some types of drivers work with additional layers of kernel support functions for a given type of device. For example, one can talk of universal serial bus (USB) modules, serial modules, SCSI modules, and so on. Every USB device is driven by a USB module that works with the USB subsystem, but the device itself shows up in the system as a char device (a USB serial port, say), a block device (a USB memory card reader), or a network device (a USB Ethernet interface).

1.4. Security Issues

Security is an increasingly important concern in modern times. We will discuss security-related issues as they come up throughout the book. There are a few general concepts, however, that are worth mentioning now.

Any security check in the system is enforced by kernel code. If the kernel has security holes, then the system as a whole has holes. In the official kernel distribution, only an authorized user can load modules; the system call init_module checks if the invoking process is authorized to load a module into the kernel. Thus, when running an official kernel, only the superuser,[1] or an intruder who has succeeded in becoming privileged, can exploit the power of privileged code.

[1] Technically, only somebody with the CAP_SYS_MODULE capability can perform this operation. We discuss capabilities in Chapter 6.

When possible, driver writers should avoid encoding security policy in their code. Security is a policy issue that is often best handled at higher levels within the kernel, under the control of the system administrator. There are always exceptions, however. As a device driver writer, you should be aware of situations in which some types of device access could adversely affect the system as a whole and should provide adequate controls. For example, device operations that affect global resources (such as setting an interrupt line), which could damage the hardware (loading firmware, for example), or that could affect other users (such as setting a default block size on a tape drive), are usually only available to sufficiently privileged users, and this check must be made in the driver itself.

Driver writers must also be careful, of course, to avoid introducing security bugs. The C programming language makes it easy to make several types of errors. Many current security problems are created, for example, by buffer overrun errors, in which the programmer forgets to check how much data is written to a buffer, and data ends up written beyond the end of the buffer, thus overwriting unrelated data. Such errors can compromise the entire system and must be avoided. Fortunately, avoiding these errors is usually relatively easy in the device driver context, in which the interface to the user is narrowly defined and highly controlled.

Some other general security ideas are worth keeping in mind. Any input received from user processes should be treated with great suspicion; never trust it unless you can verify it. Be careful with uninitialized memory; any memory obtained from the kernel should be zeroed or otherwise initialized before being made available to a user process or device. Otherwise, information leakage (disclosure of data, passwords, etc.) could result. If your device interprets data sent to it, be sure the user cannot send anything that could compromise the system. Finally, think about the possible effect of device operations; if there are specific operations (e.g., reloading the firmware on an adapter board or formatting a disk) that could affect the system, those operations should almost certainly be restricted to privileged users.

Be careful, also, when receiving software from third parties, especially when the kernel is concerned: because everybody has access to the source code, everybody can break and recompile things. Although you can usually trust precompiled kernels found in your distribution, you should avoid running kernels compiled by an untrusted friend—if you wouldn't run a precompiled binary as root, then you'd better not run a precompiled kernel. For example, a maliciously modified kernel could allow anyone to load a module, thus opening an unexpected back door via init_module.

Note that the Linux kernel can be compiled to have no module support whatsoever, thus closing any module-related security holes. In this case, of course, all needed drivers must be built directly into the kernel itself. It is also possible, with 2.2 and later kernels, to disable the loading of kernel modules after system boot via the capability mechanism.

1.5. Version Numbering

Before digging into programming, we should comment on the version numbering scheme used in Linux and which versions are covered by this book.

First of all, note that every software package used in a Linux system has its own release number, and there are often interdependencies across them: you need a particular version of one package to run a particular version of another package. The creators of Linux distributions usually handle the messy problem of matching packages, and the user who installs from a prepackaged distribution doesn't need to deal with version numbers. Those who replace and upgrade system software, on the other hand, are on their own in this regard. Fortunately, almost all modern distributions support the upgrade of single packages by checking interpackage dependencies; the distribution's package manager generally does not allow an upgrade until the dependencies are satisfied.

To run the examples we introduce during the discussion, you won't need particular versions of any tool beyond what the 2.6 kernel requires; any recent Linux distribution can be used to run our examples. We won't detail specific requirements, because the file *Documentation/Changes* in your kernel sources is the best source of such information if you experience any problems.

As far as the kernel is concerned, the even-numbered kernel versions (i.e., 2.6.x) are the stable ones that are intended for general distribution. The odd versions (such as 2.7.x), on the contrary, are development snapshots and are quite ephemeral; the latest of them represents the current status of development, but becomes obsolete in a few days or so.

This book covers Version 2.6 of the kernel. Our focus has been to show all the features available to device driver writers in 2.6.10, the current version at the time we are writing. This edition of the book does not cover prior versions of the kernel. For those of you who are interested, the second edition covered Versions 2.0 through 2.4 in detail. That edition is still available online at http://lwn.net/Kernel/LDD2/.

Kernel programmers should be aware that the development process changed with 2.6. The 2.6 series is now accepting changes that previously would have been considered too large for a "stable" kernel. Among other things, that means that internal kernel programming interfaces can change, thus potentially obsoleting parts of this book; for this reason, the sample code accompanying the text is known to work with 2.6.10, but some modules don't compile under earlier versions. Programmers wanting to keep up with kernel programming changes are encouraged to join the mailing lists and to make use of the web sites listed in the bibliography. There is also a web page maintained at http://lwn.net/Articles/2.6-kernel-api/, which contains information about API changes that have happened since this book was published.

This text doesn't talk specifically about odd-numbered kernel versions. General users never have a reason to run development kernels. Developers experimenting with new features, however, want to be running the latest development release. They usually keep upgrading to the most recent version to pick up bug fixes and new implementations of features. Note, however, that there's no guarantee on experimental kernels,[2] and nobody helps you if you have problems due to a bug in a noncurrent odd-numbered kernel. Those who run odd-numbered versions of the kernel are usually skilled enough to dig in the code without the need for a textbook, which is another reason why we don't talk about development kernels here.

[2] Note that there's no guarantee on even-numbered kernels as well, unless you rely on a commercial provider that grants its own warranty.

Another feature of Linux is that it is a platform-independent operating system, not just "a Unix clone for PC clones" anymore: it currently supports some 20 architectures. This book is platform independent as far as possible, and all the code samples have been tested on at least the x86 and x86-64 platforms. Because the code has been tested on both 32-bit and 64-bit processors, it should compile and run on all other platforms. As you might expect, the code samples that rely on particular hardware don't work on all the supported platforms, but this is always stated in the source code.

1.6. License Terms

Linux is licensed under Version 2 of the GNU General Public License (GPL), a document devised for the GNU project by the Free Software Foundation. The GPL allows anybody to redistribute, and even sell, a product covered by the GPL, as long as the recipient has access to the source and is able to exercise the same rights. Additionally, any software product derived from a product covered by the GPL must, if it is redistributed at all, be released under the GPL.

The main goal of such a license is to allow the growth of knowledge by permitting everybody to modify programs at will; at the same time, people selling software to the public can still do their job. Despite this simple objective, there's a never-ending discussion about the GPL and its use. If you want to read the license, you can find it in several places in your system, including the top directory of your kernel source tree in the *COPYING* file.

Vendors often ask whether they can distribute kernel modules in binary form only. The answer to that question has been deliberately left ambiguous. Distribution of binary modules—as long as they adhere to the published kernel interface—has been tolerated so far. But the copyrights on the kernel are held by many developers, and not all of them agree that kernel modules are not derived products. If you or your employer wish to distribute kernel modules under a nonfree license, you really need to discuss the situation with your legal counsel. Please note also that the kernel developers have no qualms against breaking binary modules between kernel releases, even in the middle of a stable kernel series. If it is at all possible, both you and your users are better off if you release your module as free software.

If you want your code to go into the mainline kernel, or if your code requires patches to the kernel, you must use a GPL-compatible license as soon as you release the code. Although personal use of your changes doesn't force the GPL on you, if you distribute your code, you must include the source code in the distribution—people acquiring your package must be allowed to rebuild the binary at will.

As far as this book is concerned, most of the code is freely redistributable, either in source or binary form, and neither we nor O'Reilly retain any right on any derived works. All the programs are available at *ftp://ftp.ora.com/pub/examples/linux/drivers/*, and the exact license terms are stated in the *LICENSE* file in the same directory.

1.7. Joining the Kernel Development Community

As you begin writing modules for the Linux kernel, you become part of a larger community of developers. Within that community, you can find not only people engaged in similar work, but also a group of highly committed engineers working toward making Linux a better system. These people can be a source of help, ideas, and critical review as well—they will be the first people you will likely turn to when you are looking for testers for a new driver.

The central gathering point for Linux kernel developers is the linux-kernel mailing list. All major kernel developers, from Linus Torvalds on down, subscribe to this list. Please note that the list is not for the faint of heart: traffic as of this writing can run up to 200 messages per day or more. Nonetheless, following this list is essential for those who are interested in kernel development; it also can be a top-quality resource for those in need of kernel development help.

To join the linux-kernel list, follow the instructions found in the linux-kernel mailing list FAQ: *http://www.tux.org/lkml*. Read the rest of the FAQ while you are at it; there is a great deal of useful information there. Linux kernel developers are busy people, and they are much more inclined to help people who have clearly done their homework first.

1.8. Overview of the Book

From here on, we enter the world of kernel programming. Chapter 2 introduces modularization, explaining the secrets of the art and showing the code for running modules. Chapter 3 talks about char drivers and shows the complete code for a memory-based device driver that can be read and written for fun. Using memory as the hardware base for the device allows anyone to run the sample code without the need to acquire special hardware.

Debugging techniques are vital tools for the programmer and are introduced in Chapter 4. Equally important for those who would hack on contemporary kernels is the management of concurrency and race conditions. Chapter 5 concerns itself with the problems posed by concurrent access to resources and introduces the Linux mechanisms for controlling concurrency.

With debugging and concurrency management skills in place, we move to advanced features of char drivers, such as blocking operations, the use of select, and the important ioctl call; these topics are the subject of Chapter 6.

Before dealing with hardware management, we dissect a few more of the kernel's software interfaces: Chapter 7 shows how time is managed in the kernel, and Chapter 8 explains memory allocation.

Next we focus on hardware. Chapter 9 describes the management of I/O ports and memory buffers that live on the device; after that comes interrupt handling, in Chapter 10. Unfortunately, not everyone is able to run the sample code for these chapters, because some hardware support is actually needed to test the software interface interrupts. We've tried our best to keep required hardware support to a minimum, but you still need some simple hardware, such as a standard parallel port, to work with the sample code for these chapters.

Chapter 11 covers the use of data types in the kernel and the writing of portable code.

The second half of the book is dedicated to more advanced topics. We start by getting deeper into the hardware and, in particular, the functioning of specific peripheral buses. Chapter 12 covers the details of writing drivers for PCI devices, and Chapter 13 examines the API for working with USB devices.

With an understanding of peripheral buses in place, we can take a detailed look at the Linux device model, which is the abstraction layer used by the kernel to describe the hardware and software resources it is managing. Chapter 14 is a bottom-up look at the device model infrastructure, starting with the kobject type and working up from there. It covers the integration of the device model with real hardware; it then uses that knowledge to cover topics like hot-pluggable devices and power management.

In Chapter 15, we take a diversion into Linux memory management. This chapter shows how to map kernel memory into user space (the mmap system call), map user memory into kernel space (with get_user_pages), and how to map either kind of memory into device space (to perform direct memory access [DMA] operations).

Our understanding of memory will be useful for the following two chapters, which cover the other major driver classes. Chapter 16 introduces block drivers and shows how they are different from the char drivers we have worked with so far. Then Chapter 17 gets into the writing of network drivers. We finish up with a discussion of serial drivers and a bibliography.

Chapter 2. Building and Running Modules

It's almost time to begin programming. This chapter introduces all the essential concepts about modules and kernel programming. In these few pages, we build and run a complete (if relatively useless) module, and look at some of the basic code shared by all modules. Developing such expertise is an essential foundation for any kind of modularized driver. To avoid throwing in too many concepts at once, this chapter talks only about modules, without referring to any specific device class.

All the kernel items (functions, variables, header files, and macros) that are introduced here are described in a reference section at the end of the chapter.

2.1. Setting Up Your Test System

Starting with this chapter, we present example modules to demonstrate programming concepts. (All of these examples are available on O'Reilly's FTP site, as explained in Chapter 1.) Building, loading, and modifying these examples are a good way to improve your understanding of how drivers work and interact with the kernel.

The example modules should work with almost any 2.6.x kernel, including those provided by distribution vendors. However, we recommend that you obtain a "mainline" kernel directly from the kernel.org mirror network, and install it on your system. Vendor kernels can be heavily patched and divergent from the mainline; at times, vendor patches can change the kernel API as seen by device drivers. If you are writing a driver that must work on a particular distribution, you will certainly want to build and test against the relevant kernels. But, for the purpose of learning about driver writing, a standard kernel is best.

Regardless of the origin of your kernel, building modules for 2.6.x requires that you have a configured and built kernel tree on your system. This requirement is a change from previous versions of the kernel, where a current set of header files was sufficient. 2.6 modules are linked against object files found in the kernel source tree; the result is a more robust module loader, but also the requirement that those object files be available. So your first order of business is to come up with a kernel source tree (either from the kernel.org network or your distributor's kernel source package), build a new kernel, and install it on your system. For reasons we'll see later, life is generally easiest if you are actually running the target kernel when you build your modules, though this is not required.

You should also give some thought to where you do your module experimentation, development, and testing. We have done our best to make our example modules safe and correct, but the possibility of bugs is always present. Faults in kernel code can bring about the demise of a user process or, occasionally, the entire system. They do not normally create more serious problems, such as disk corruption. Nonetheless, it is advisable to do your kernel experimentation on a system that does not contain data that you cannot afford to lose, and that does not perform essential services. Kernel hackers typically keep a "sacrificial" system around for the purpose of testing new code.

So, if you do not yet have a suitable system with a configured and built kernel source tree on disk, now would be a good time to set that up. We'll wait. Once that task is taken care of, you'll be ready to start playing with kernel modules.

2.2. The Hello World Module

Many programming books begin with a "hello world" example as a way of showing the simplest possible program. This book deals in kernel modules rather than programs; so, for the impatient reader, the following code is a complete "hello world" module:

```
#include <linux/init.h>

#include <linux/module.h>

MODULE_LICENSE("Dual BSD/GPL");



static int hello_init(void)

{

    printk(KERN_ALERT "Hello, world\n");

    return 0;

}



static void hello_exit(void)

{

    printk(KERN_ALERT "Goodbye, cruel world\n");

}



module_init(hello_init);

module_exit(hello_exit);
```

This module defines two functions, one to be invoked when the module is loaded into the kernel (hello_init) and one for when the module is removed (hello_exit). The module_init and module_exit lines use special kernel macros to indicate the role of these two functions. Another special macro (MODULE_LICENSE) is used to tell the kernel that this module bears a free license; without such a declaration, the kernel complains when the module is loaded.

The printk function is defined in the Linux kernel and made available to modules; it behaves similarly to the standard C library function printf. The kernel needs its own printing function because it runs by itself, without the help of the C library. The module can call printk because, after insmod has loaded it, the module is linked to the kernel and can access the kernel's public symbols (functions and variables, as detailed in the next section). The string KERN_ALERT is the priority of the message.[1] We've specified a high priority in this module, because a message with the default priority might not show up anywhere useful, depending on the kernel version you are running, the version of the klogd daemon, and your configuration. You can ignore this issue for now; we explain it in Chapter 4.

[1] The priority is just a string, such as <1>, which is prepended to the printk format string. Note the lack of a comma after KERN_ALERT; adding a comma there is a common and annoying typo (which, fortunately, is caught by the compiler).

You can test the module with the insmod and rmmod utilities, as shown below. Note that only the superuser can load and unload a module.

```
% make

make[1]: Entering directory `/usr/src/linux-2.6.10'
```

2.3. Kernel Modules Versus Applications

Before we go further, it's worth underlining the various differences between a kernel module and an application.

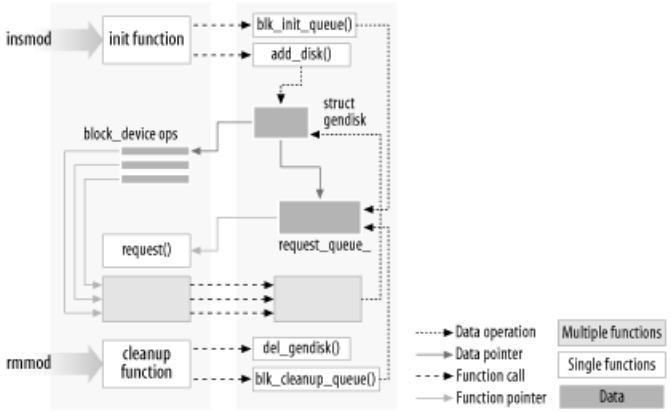
While most small and medium-sized applications perform a single task from beginning to end, every kernel module just registers itself in order to serve future requests, and its initialization function terminates immediately. In other words, the task of the module's initialization function is to prepare for later invocation of the module's functions; it's as though the module were saying, "Here I am, and this is what I can do." The module's exit function (hello_exit in the example) gets invoked just before the module is unloaded. It should tell the kernel, "I'm not there anymore; don't ask me to do anything else." This kind of approach to programming is similar to event-driven programming, but while not all applications are event-driven, each and every kernel module is. Another major difference between event-driven applications and kernel code is in the exit function: whereas an application that terminates can be lazy in releasing resources or avoids clean up altogether, the exit function of a module must carefully undo everything the init function built up, or the pieces remain around until the system is rebooted.

Incidentally, the ability to unload a module is one of the features of modularization that you'll most appreciate, because it helps cut down development time; you can test successive versions of your new driver without going through the lengthy shutdown/reboot cycle each time.

As a programmer, you know that an application can call functions it doesn't define: the linking stage resolves external references using the appropriate library of functions. printf is one of those callable functions and is defined in libc. A module, on the other hand, is linked only to the kernel, and the only functions it can call are the ones exported by the kernel; there are no libraries to link to. The printk function used in *hello.c* earlier, for example, is the version of printf defined within the kernel and exported to modules. It behaves similarly to the original function, with a few minor differences, the main one being lack of floating-point support.

Figure 2-1 shows how function calls and function pointers are used in a module to add new functionality to a running kernel.

Figure 2-1. Linking a module to the kernel



Because no library is linked to modules, source files should never include the usual header files, *<stdarg.h>* and very special situations being the only exceptions. Only functions that are actually part of the kernel itself may be used in kernel modules. Anything related to the kernel is declared in headers found in the kernel source tree you have set up and configured; most of the relevant headers live in *include/linux* and *include/asm*, but other subdirectories of *include* have been added to host material associated to specific kernel subsystems.

The role of individual kernel headers is introduced throughout the book as each of them is needed.

Another important difference between kernel programming and application programming is in how each environment handles faults: whereas a segmentation fault is harmless during application development and a debugger can always be used to trace the error to the problem in the source code, a kernel fault kills the current process at least, if not the whole system. We see how to trace kernel errors in Chapter 4.

2.4. Compiling and Loading

The "hello world" example at the beginning of this chapter included a brief demonstration of building a module and loading it into the system. There is, of course, a lot more to that whole process than we have seen so far. This section provides more detail on how a module author turns source code into an executing subsystem within the kernel.

2.4.1. Compiling Modules

As the first step, we need to look a bit at how modules must be built. The build process for modules differs significantly from that used for user-space applications; the kernel is a large, standalone program with detailed and explicit requirements on how its pieces are put together. The build process also differs from how things were done with previous versions of the kernel; the new build system is simpler to use and produces more correct results, but it looks very different from what came before. The kernel build system is a complex beast, and we just look at a tiny piece of it. The files found in the *Documentation/kbuild* directory in the kernel source are required reading for anybody wanting to understand all that is really going on beneath the surface.

There are some prerequisites that you must get out of the way before you can build kernel modules. The first is to ensure that you have sufficiently current versions of the compiler, module utilities, and other necessary tools. The file *Documentation/Changes* in the kernel documentation directory always lists the required tool versions; you should consult it before going any further. Trying to build a kernel (and its modules) with the wrong tool versions can lead to no end of subtle, difficult problems. Note that, occasionally, a version of the compiler that is too new can be just as problematic as one that is too old; the kernel source makes a great many assumptions about the compiler, and new releases can sometimes break things for a while.

If you still do not have a kernel tree handy, or have not yet configured and built that kernel, now is the time to go do it. You cannot build loadable modules for a 2.6 kernel without this tree on your filesystem. It is also helpful (though not required) to be actually running the kernel that you are building for.

Once you have everything set up, creating a makefile for your module is straightforward. In fact, for the "hello world" example shown earlier in this chapter, a single line will suffice:

```
obj-m := hello.o
```

Readers who are familiar with make, but not with the 2.6 kernel build system, are likely to be wondering how this makefile works. The above line is not how a traditional makefile looks, after all. The answer, of course, is that the kernel build system handles the rest. The assignment above (which takes advantage of the extended syntax provided by GNU make) states that there is one module to be built from the object file *hello.o*. The resulting module is named *hello.ko* after being built from the object file.

If, instead, you have a module called *module.ko* that is generated from two source files (called, say, *file1.c* and *file2.c*), the correct incantation would be:

```
obj-m := module.o

module-objs := file1.o file2.o
```

For a makefile like those shown above to work, it must be invoked within the context of the larger kernel build system. If your kernel source tree is located in, say, your *~/kernel-2.6* directory, the make command required to build your module (typed in the directory containing the module source and makefile) would be:

```
make -C ~/kernel-2.6 M=`pwd` modules
```

This command starts by changing its directory to the one provided with the -C option (that is, your kernel source directory). There it finds the kernel's top-level makefile. The M= option causes that makefile to move back into your module source directory before trying to build the modules target. This target, in turn, refers to the list of modules found in the obj-m variable, which we've set to *module.o* in our examples.

Typing the previous make command can get tiresome after a while, so the kernel developers have developed a sort of makefile idiom, which makes life easier for those building modules outside of the kernel tree. The trick is to write your makefile as follows:

```
# If KERNELRELEASE is defined, we've been invoked from the

# kernel build system and can use its language.
```

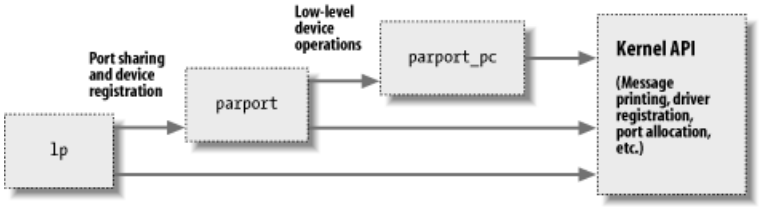
2.5. The Kernel Symbol Table

We've seen how insmod resolves undefined symbols against the table of public kernel symbols. The table contains the addresses of global kernel items—functions and variables—that are needed to implement modularized drivers. When a module is loaded, any symbol exported by the module becomes part of the kernel symbol table. In the usual case, a module implements its own functionality without the need to export any symbols at all. You need to export symbols, however, whenever other modules may benefit from using them.

New modules can use symbols exported by your module, and you can stack new modules on top of other modules. Module stacking is implemented in the mainstream kernel sources as well: the msdos filesystem relies on symbols exported by the fat module, and each input USB device module stacks on the usbcore and input modules.

Module stacking is useful in complex projects. If a new abstraction is implemented in the form of a device driver, it might offer a plug for hardware-specific implementations. For example, the video-for-linux set of drivers is split into a generic module that exports symbols used by lower-level device drivers for specific hardware. According to your setup, you load the generic video module and the specific module for your installed hardware. Support for parallel ports and the wide variety of attachable devices is handled in the same way, as is the USB kernel subsystem. Stacking in the parallel port subsystem is shown in Figure 2-2; the arrows show the communications between the modules and with the kernel programming interface.

Figure 2-2. Stacking of parallel port driver modules



When using stacked modules, it is helpful to be aware of the modprobe utility. As we described earlier, modprobe functions in much the same way as insmod, but it also loads any other modules that are required by the module you want to load. Thus, one modprobe command can sometimes replace several invocations of insmod (although you'll still need insmod when loading your own modules from the current directory, because modprobe looks only in the standard installed module directories).

Using stacking to split modules into multiple layers can help reduce development time by simplifying each layer. This is similar to the separation between mechanism and policy that we discussed in Chapter 1.

The Linux kernel header files provide a convenient way to manage the visibility of your symbols, thus reducing namespace pollution (filling the namespace with names that may conflict with those defined elsewhere in the kernel) and promoting proper information hiding. If your module needs to export symbols for other modules to use, the following macros should be used.

```
EXPORT_SYMBOL(name);

EXPORT_SYMBOL_GPL(name);
```

Either of the above macros makes the given symbol available outside the module. The _GPL version makes the symbol available to GPL-licensed modules only. Symbols must be exported in the global part of the module's file, outside of any function, because the macros expand to the declaration of a special-purpose variable that is expected to be accessible globally. This variable is stored in a special part of the module executible (an "ELF section") that is used by the kernel at load time to find the variables exported by the module. (Interested readers can look at *<linux/module.h>* for the details, even though the details are not needed to make things work.)

2.6. Preliminaries

We are getting closer to looking at some actual module code. But first, we need to look at some other things that need to appear in your module source files. The kernel is a unique environment, and it imposes its own requirements on code that would interface with it.

Most kernel code ends up including a fairly large number of header files to get definitions of functions, data types, and variables. We'll examine these files as we come to them, but there are a few that are specific to modules, and must appear in every loadable module. Thus, just about all module code has the following:

```
#include <linux/module.h>

#include <linux/init.h>
```

module.h contains a great many definitions of symbols and functions needed by loadable modules. You need *init.h* to specify your initialization and cleanup functions, as we saw in the "hello world" example above, and which we revisit in the next section. Most modules also include *moduleparam.h* to enable the passing of parameters to the module at load time; we will get to that shortly.

It is not strictly necessary, but your module really should specify which license applies to its code. Doing so is just a matter of including a MODULE_LICENSE line:

```
MODULE_LICENSE("GPL");
```

The specific licenses recognized by the kernel are "GPL" (for any version of the GNU General Public License), "GPL v2" (for GPL version two only), "GPL and additional rights," "Dual BSD/GPL," "Dual MPL/GPL," and "Proprietary." Unless your module is explicitly marked as being under a free license recognized by the kernel, it is assumed to be proprietary, and the kernel is "tainted" when the module is loaded. As we mentioned in Section 1.6, kernel developers tend to be unenthusiastic about helping users who experience problems after loading proprietary modules.

Other descriptive definitions that can be contained within a module include MODULE_AUTHOR (stating who wrote the module), MODULE_DESCRIPTION (a human-readable statement of what the module does), MODULE_VERSION (for a code revision number; see the comments in *<linux/module.h>* for the conventions to use in creating version strings), MODULE_ALIAS (another name by which this module can be known), and MODULE_DEVICE_TABLE (to tell user space about which devices the module supports).

The various MODULE_ declarations can appear anywhere within your source file outside of a function. A relatively recent convention in kernel code, however, is to put these declarations at the end of the file.

2.7. Initialization and Shutdown

As already mentioned, the module initialization function registers any facility offered by the module. By facility, we mean a new functionality, be it a whole driver or a new software abstraction, that can be accessed by an application. The actual definition of the initialization function always looks like:

```
static int __init initialization_function(void)

{

    /* Initialization code here */

}

module_init(initialization_function);
```

Initialization functions should be declared static, since they are not meant to be visible outside the specific file; there is no hard rule about this, though, as no function is exported to the rest of the kernel unless explicitly requested. The __init token in the definition may look a little strange; it is a hint to the kernel that the given function is used only at initialization time. The module loader drops the initialization function after the module is loaded, making its memory available for other uses. There is a similar tag (__initdata) for data used only during initialization. Use of __init and __initdata is optional, but it is worth the trouble. Just be sure not to use them for any function (or data structure) you will be using after initialization completes. You may also encounter __devinit and __devinitdata in the kernel source; these translate to __init and __initdata only if the kernel has not been configured for hotpluggable devices. We will look at hotplug support in Chapter 14.

The use of module_init is mandatory. This macro adds a special section to the module's object code stating where the module's initialization function is to be found. Without this definition, your initialization function is never called.

Modules can register many different types of facilities, including different kinds of devices, filesystems, cryptographic transforms, and more. For each facility, there is a specific kernel function that accomplishes this registration. The arguments passed to the kernel registration functions are usually pointers to data structures describing the new facility and the name of the facility being registered. The data structure usually contains pointers to module functions, which is how functions in the module body get called.

The items that can be registered go beyond the list of device types mentioned in Chapter 1. They include, among others, serial ports, miscellaneous devices, sysfs entries, */proc* files, executable domains, and line disciplines. Many of those registrable items support functions that aren't directly related to hardware but remain in the "software abstractions" field. Those items can be registered, because they are integrated into the driver's functionality anyway (like */proc* files and line disciplines for example).

There are other facilities that can be registered as add-ons for certain drivers, but their use is so specific that it's not worth talking about them; they use the stacking technique, as described in Section 2.5. If you want to probe further, you can grep for EXPORT_SYMBOL in the kernel sources, and find the entry points offered by different drivers. Most registration functions are prefixed with register_, so another possible way to find them is to grep for register_ in the kernel source.

2.7.1. The Cleanup Function

Every nontrivial module also requires a cleanup function, which unregisters interfaces and returns all resources to the system before the module is removed. This function is defined as:

```
static void __exit cleanup_function(void)

{

    /* Cleanup code here */

}



module_exit(cleanup_function);
```

2.8. Module Parameters

Several parameters that a driver needs to know can change from system to system. These can vary from the device number to use (as we'll see in the next chapter) to numerous aspects of how the driver should operate. For example, drivers for SCSI adapters often have options controlling the use of tagged command queuing, and the Integrated Device Electronics (IDE) drivers allow user control of DMA operations. If your driver controls older hardware, it may also need to be told explicitly where to find that hardware's I/O ports or I/O memory addresses. The kernel supports these needs by making it possible for a driver to designate parameters that may be changed when the driver's module is loaded.

These parameter values can be assigned at load time by insmod or modprobe ; the latter can also read parameter assignment from its configuration file (*/etc/modprobe.conf*). The commands accept the specification of several types of values on the command line. As a way of demonstrating this capability, imagine a much-needed enhancement to the "hello world" module (called hellop) shown at the beginning of this chapter. We add two parameters: an integer value called howmany and a character string called whom. Our vastly more functional module then, at load time, greets whom not just once, but howmany times. Such a module could then be loaded with a command line such as:

```
insmod hellop howmany=10 whom="Mom"
```

Upon being loaded that way, hellop would say "Hello, Mom" 10 times.

However, before insmod can change module parameters, the module must make them available. Parameters are declared with the module_param macro, which is defined in *moduleparam.h*. module_param takes three parameters: the name of the variable, its type, and a permissions mask to be used for an accompanying sysfs entry. The macro should be placed outside of any function and is typically found near the head of the source file. So hellop would declare its parameters and make them available to insmod as follows:

```
static char *whom = "world";

static int howmany = 1;

module_param(howmany, int, S_IRUGO);

module_param(whom, charp, S_IRUGO);
```

Numerous types are supported for module parameters:

bool

invbool

A boolean (true or false) value (the associated variable should be of type int). The invbool type inverts the value, so that true values become false and vice versa.

charp

A char pointer value. Memory is allocated for user-provided strings, and the pointer is set accordingly.

int

long

short

uint

2.9. Doing It in User Space

A Unix programmer who's addressing kernel issues for the first time might be nervous about writing a module. Writing a user program that reads and writes directly to the device ports may be easier.

Indeed, there are some arguments in favor of user-space programming, and sometimes writing a so-called user-space device driver is a wise alternative to kernel hacking. In this section, we discuss some of the reasons why you might write a driver in user space. This book is about kernel-space drivers, however, so we do not go beyond this introductory discussion.

The advantages of user-space drivers are:

- The full C library can be linked in. The driver can perform many exotic tasks without resorting to external programs (the utility programs implementing usage policies that are usually distributed along with the driver itself).
- The programmer can run a conventional debugger on the driver code without having to go through contortions to debug a running kernel.
- If a user-space driver hangs, you can simply kill it. Problems with the driver are unlikely to hang the entire system, unless the hardware being controlled is really misbehaving.
- User memory is swappable, unlike kernel memory. An infrequently used device with a huge driver won't occupy RAM that other programs could be using, except when it is actually in use.
- A well-designed driver program can still, like kernel-space drivers, allow concurrent access to a device.
- If you must write a closed-source driver, the user-space option makes it easier for you to avoid ambiguous licensing situations and problems with changing kernel interfaces.

For example, USB drivers can be written for user space; see the (still young) libusb project at libusb.sourceforge.net and "gadgetfs" in the kernel source. Another example is the X server: it knows exactly what the hardware can do and what it can't, and it offers the graphic resources to all X clients. Note, however, that there is a slow but steady drift toward frame-buffer-based graphics environments, where the X server acts only as a server based on a real kernel-space device driver for actual graphic manipulation.

Usually, the writer of a user-space driver implements a server process, taking over from the kernel the task of being the single agent in charge of hardware control. Client applications can then connect to the server to perform actual communication with the device; therefore, a smart driver process can allow concurrent access to the device. This is exactly how the X server works.

But the user-space approach to device driving has a number of drawbacks. The most important are:

- Interrupts are not available in user space. There are workarounds for this limitation on some platforms, such as the vm86 system call on the IA32 architecture.
- Direct access to memory is possible only by mmapping */dev/mem*, and only a privileged user can do that.
- Access to I/O ports is available only after calling ioperm or iopl. Moreover, not all platforms support these system calls, and access to */dev/port* can be too slow to be effective. Both the system calls and the

2.10. Quick Reference

This section summarizes the kernel functions, variables, macros, and */proc* files that we've touched on in this chapter. It is meant to act as a reference. Each item is listed after the relevant header file, if any. A similar section appears at the end of almost every chapter from here on, summarizing the new symbols introduced in the chapter. Entries in this section generally appear in the same order in which they were introduced in the chapter:

insmod

modprobe

rmmod

User-space utilities that load modules into the running kernels and remove them.

#include <linux/init.h>

module_init(init_function);

module_exit(cleanup_function);

Macros that designate a module's initialization and cleanup functions.

__init

__initdata

__exit

__exitdata

Markers for functions (__init and __exit) and data (__initdata and __exitdata) that are only used at module initialization or cleanup time. Items marked for initialization may be discarded once initialization completes; the exit items may be discarded if module unloading has not been configured into the kernel. These markers work by causing the relevant objects to be placed in a special ELF section in the executable file.

#include <linux/sched.h>

One of the most important header files. This file contains definitions of much of the kernel API used by the driver, including functions for sleeping and numerous variable declarations.

struct task_struct *current;

The current process.

current->pid

Chapter 3. Char Drivers

The goal of this chapter is to write a complete char device driver. We develop a character driver because this class is suitable for most simple hardware devices. Char drivers are also easier to understand than block drivers or network drivers (which we get to in later chapters). Our ultimate aim is to write a modularized char driver, but we won't talk about modularization issues in this chapter.

Throughout the chapter, we present code fragments extracted from a real device driver: scull (Simple Character Utility for Loading Localities). scull is a char driver that acts on a memory area as though it were a device. In this chapter, because of that peculiarity of scull, we use the word device interchangeably with "the memory area used by scull."

The advantage of scull is that it isn't hardware dependent. scull just acts on some memory, allocated from the kernel. Anyone can compile and run scull, and scull is portable across the computer architectures on which Linux runs. On the other hand, the device doesn't do anything "useful" other than demonstrate the interface between the kernel and char drivers and allow the user to run some tests.

3.1. The Design of scull

The first step of driver writing is defining the capabilities (the mechanism) the driver will offer to user programs. Since our "device" is part of the computer's memory, we're free to do what we want with it. It can be a sequential or random-access device, one device or many, and so on.

To make scull useful as a template for writing real drivers for real devices, we'll show you how to implement several device abstractions on top of the computer memory, each with a different personality.

The scull source implements the following devices. Each kind of device implemented by the module is referred to as a type .

scull0 to scull3

Four devices, each consisting of a memory area that is both global and persistent. Global means that if the device is opened multiple times, the data contained within the device is shared by all the file descriptors that opened it. Persistent means that if the device is closed and reopened, data isn't lost. This device can be fun to work with, because it can be accessed and tested using conventional commands, such as cp, cat, and shell I/O redirection.

scullpipe0 to scullpipe3

Four FIFO (first-in-first-out) devices, which act like pipes. One process reads what another process writes. If multiple processes read the same device, they contend for data. The internals of scullpipe will show how blocking and nonblocking read and write can be implemented without having to resort to interrupts. Although real drivers synchronize with their devices using hardware interrupts, the topic of blocking and nonblocking operations is an important one and is separate from interrupt handling (covered in Chapter 10).

scullsingle

scullpriv

sculluid

scullwuid

These devices are similar to scull0 but with some limitations on when an open is permitted. The first (scullsingle) allows only one process at a time to use the driver, whereas scullpriv is private to each virtual console (or X terminal session), because processes on each console/terminal get different memory areas. sculluid and scullwuid can be opened multiple times, but only by one user at a time; the former returns an error of "Device Busy" if another user is locking the device, whereas the latter implements blocking open. These variations of scull would appear to be confusing policy and mechanism, but they are worth looking at, because some real-life devices require this sort of management.

Each of the scull devices demonstrates different features of a driver and presents different difficulties. This chapter covers the internals of scull0 to scull3; the more advanced devices are covered in Chapter 6. scullpipe is described in the section Section 3.4 and the others are described in Section 6.6

3.2. Major and Minor Numbers

Char devices are accessed through names in the filesystem. Those names are called special files or device files or simply nodes of the filesystem tree; they are conventionally located in the */dev* directory. Special files for char drivers are identified by a "c" in the first column of the output of ls -l. Block devices appear in */dev* as well, but they are identified by a "b." The focus of this chapter is on char devices, but much of the following information applies to block devices as well.

If you issue the ls -l command, you'll see two numbers (separated by a comma) in the device file entries before the date of the last modification, where the file length normally appears. These numbers are the major and minor device number for the particular device. The following listing shows a few devices as they appear on a typical system. Their major numbers are 1, 4, 7, and 10, while the minors are 1, 3, 5, 64, 65, and 129.

```
  crw-rw-rw-    1 root     root       1,   3 Apr 11  2002 null

 crw-------    1 root     root      10,   1 Apr 11  2002 psaux

 crw-------    1 root     root       4,   1 Oct 28 03:04 tty1

 crw-rw-rw-    1 root     tty        4,  64 Apr 11  2002 ttys0

 crw-rw----    1 root     uucp       4,  65 Apr 11  2002 ttyS1

 crw--w----    1 vcsa     tty        7,   1 Apr 11  2002 vcs1

 crw--w----    1 vcsa     tty        7, 129 Apr 11  2002 vcsa1

 crw-rw-rw-    1 root     root       1,   5 Apr 11  2002 zero
```

Traditionally, the major number identifies the driver associated with the device. For example, */dev/null* and */dev/zero* are both managed by driver 1, whereas virtual consoles and serial terminals are managed by driver 4; similarly, both *vcs1* and *vcsa1* devices are managed by driver 7. Modern Linux kernels allow multiple drivers to share major numbers, but most devices that you will see are still organized on the one-major-one-driver principle.

The minor number is used by the kernel to determine exactly which device is being referred to. Depending on how your driver is written (as we will see below), you can either get a direct pointer to your device from the kernel, or you can use the minor number yourself as an index into a local array of devices. Either way, the kernel itself knows almost nothing about minor numbers beyond the fact that they refer to devices implemented by your driver.

3.2.1. The Internal Representation of Device Numbers

Within the kernel, the dev_t type (defined in <*linux/types.h*>) is used to hold device numbers—both the major and minor parts. As of Version 2.6.0 of the kernel, dev_t is a 32-bit quantity with 12 bits set aside for the major number and 20 for the minor number. Your code should, of course, never make any assumptions about the internal organization of device numbers; it should, instead, make use of a set of macros found in <*linux/kdev_t.h*>. To obtain the major or minor parts of a dev_t, use:

```
 MAJOR(dev_t dev);

MINOR(dev_t dev);
```

If, instead, you have the major and minor numbers and need to turn them into a dev_t, use:

```
 MKDEV(int major, int minor);
```

Note that the 2.6 kernel can accommodate a vast number of devices, while previous kernel versions were limited to 255 major and 255 minor numbers. One assumes that the wider range will be sufficient for quite some time, but the computing field is littered with erroneous assumptions of that nature. So you should expect that the format of dev_t could change again in the future; if you write your drivers carefully, however, these changes will not be a problem.

3.2.2. Allocating and Freeing Device Numbers

3.3. Some Important Data Structures

As you can imagine, device number registration is just the first of many tasks that driver code must carry out. We will soon look at other important driver components, but one other digression is needed first. Most of the fundamental driver operations involve three important kernel data structures, called file_operations, file, and inode. A basic familiarity with these structures is required to be able to do much of anything interesting, so we will now take a quick look at each of them before getting into the details of how to implement the fundamental driver operations.

3.3.1. File Operations

So far, we have reserved som e device numbers for our use, but we have not yet connected any of our driver's operations to those numbers. The file_operations structure is how a char driver sets up this connection. The structure, defined in *<linux/fs.h>*, is a collection of function pointers. Each open file (represented internally by a file structure, which we will examine shortly) is associated with its own set of functions (by including a field called f_op that points to a file_operations structure). The operations are mostly in charge of implementing the system calls and are therefore, named open, read, and so on. We can consider the file to be an "object" and the functions operating on it to be its "methods," using object-oriented programming terminology to denote actions declared by an object to act on itself. This is the first sign of object-oriented programming we see in the Linux kernel, and we'll see more in later chapters.

Conventionally, a file_operations structure or a pointer to one is called fops (or some variation thereof). Each field in the structure must point to the function in the driver that implements a specific operation, or be left NULL for unsupported operations. The exact behavior of the kernel when a NULL pointer is specified is different for each function, as the list later in this section shows.

The following list introduces all the operations that an application can invoke on a device. We've tried to keep the list brief so it can be used as a reference, merely summarizing each operation and the default kernel behavior when a NULL pointer is used.

As you read through the list of file_operations methods, you will note that a number of parameters include the string _ _user. This annotation is a form of documentation, noting that a pointer is a user-space address that cannot be directly dereferenced. For normal compilation, _ _user has no effect, but it can be used by external checking software to find misuse of user-space addresses.

The rest of the chapter, after describing some other important data structures, explains the role of the most important operations and offers hints, caveats, and real code examples. We defer discussion of the more complex operations to later chapters, because we aren't ready to dig into topics such as memory management, blocking operations, and asynchronous notification quite yet.

struct module *owner

The first file_operations field is not an operation at all; it is a pointer to the module that "owns" the structure. This field is used to prevent the module from being unloaded while its operations are in use. Almost all the time, it is simply initialized to THIS_MODULE, a macro defined in *<linux/module.h>*.

loff_t (*llseek) (struct file *, loff_t, int);

The llseek method is used to change the current read/write position in a file, and the new position is returned as a (positive) return value. The loff_t parameter is a "long offset" and is at least 64 bits wide even on 32-bit platforms. Errors are signaled by a negative return value. If this function pointer is NULL, seek calls will modify the position counter in the file structure (described in Section 3.3.2) in potentially unpredictable ways.

ssize_t (*read) (struct file *, char _ _user *, size_t, loff_t *);

Used to retrieve data from the device. A null pointer in this position causes the read system call to fail with -EINVAL ("Invalid argument"). A nonnegative return value represents the number of bytes successfully read (the

3.4. Char Device Registration

As we mentioned, the kernel uses structures of type struct cdev to represent char devices internally. Before the kernel invokes your device's operations, you must allocate and register one or more of these structures.[6] To do so, your code should include <*linux/cdev.h*>, where the structure and its associated helper functions are defined.

[6] There is an older mechanism that avoids the use of cdev structures (which we discuss in Section 3.4.2). New code should use the newer technique, however.

There are two ways of allocating and initializing one of these structures. If you wish to obtain a standalone cdev structure at runtime, you may do so with code such as:

```
struct cdev *my_cdev = cdev_alloc(  );

my_cdev->ops = &my_fops;
```

Chances are, however, that you will want to embed the cdev structure within a device-specific structure of your own; that is what scull does. In that case, you should initialize the structure that you have already allocated with:

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

Either way, there is one other struct cdev field that you need to initialize. Like the file_operations structure, struct cdev has an owner field that should be set to THIS_MODULE.

Once the cdev structure is set up, the final step is to tell the kernel about it with a call to:

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

Here, dev is the cdev structure, num is the first device number to which this device responds, and count is the number of device numbers that should be associated with the device. Often count is one, but there are situations where it makes sense to have more than one device number correspond to a specific device. Consider, for example, the SCSI tape driver, which allows user space to select operating modes (such as density) by assigning multiple minor numbers to each physical device.

There are a couple of important things to keep in mind when using cdev_add. The first is that this call can fail. If it returns a negative error code, your device has not been added to the system. It almost always succeeds, however, and that brings up the other point: as soon as cdev_add returns, your device is "live" and its operations can be called by the kernel. You should not call cdev_add until your driver is completely ready to handle operations on the device.

To remove a char device from the system, call:

```
void cdev_del(struct cdev *dev);
```

Clearly, you should not access the cdev structure after passing it to cdev_del.

3.4.1. Device Registration in scull

Internally, scull represents each device with a structure of type struct scull_dev. This structure is defined as:

```
struct scull_dev {

    struct scull_qset *data;  /* Pointer to first quantum set */

    int quantum;              /* the current quantum size */

    int qset;                 /* the current array size */

    unsigned long size;       /* amount of data stored here */

    unsigned int access_key;  /* used by sculluid and scullpriv */

    struct semaphore sem;     /* mutual exclusion semaphore     */

    struct cdev cdev;     /* Char device structure      */
```

3.5. open and release

Now that we've taken a quick look at the fields, we start using them in real scull functions.

3.5.1. The open Method

The open method is provided for a driver to do any initialization in preparation for later operations. In most drivers, open should perform the following tasks:

- Check for device-specific errors (such as device-not-ready or similar hardware problems)
- Initialize the device if it is being opened for the first time
- Update the f_op pointer, if necessary
- Allocate and fill any data structure to be put in filp->private_data

The first order of business, however, is usually to identify which device is being opened. Remember that the prototype for the open method is:

```
int (*open)(struct inode *inode, struct file *filp);
```

The inode argument has the information we need in the form of its i_cdev field, which contains the cdev structure we set up before. The only problem is that we do not normally want the cdev structure itself, we want the scull_dev structure that contains that cdev structure. The C language lets programmers play all sorts of tricks to make that kind of conversion; programming such tricks is error prone, however, and leads to code that is difficult for others to read and understand. Fortunately, in this case, the kernel hackers have done the tricky stuff for us, in the form of the container_of macro, defined in *<linux/kernel.h>*:

```
container_of(pointer, container_type, container_field);
```

This macro takes a pointer to a field of type container_field, within a structure of type container_type, and returns a pointer to the containing structure. In scull_open, this macro is used to find the appropriate device structure:

```
struct scull_dev *dev; /* device information */



dev = container_of(inode->i_cdev, struct scull_dev, cdev);

filp->private_data = dev; /* for other methods */
```

Once it has found the scull_dev structure, scull stores a pointer to it in the private_data field of the file structure for easier access in the future.

The other way to identify the device being opened is to look at the minor number stored in the inode structure. If you register your device with register_chrdev, you must use this technique. Be sure to use iminor to obtain the minor number from the inode structure, and make sure that it corresponds to a device that your driver is actually prepared to handle.

The (slightly simplified) code for scull_open is:

```
int scull_open(struct inode *inode, struct file *filp)

{

    struct scull_dev *dev; /* device information */
```

3.6. scull's Memory Usage

Before introducing the read and write operations, we'd better look at how and why scull performs memory allocation. "How" is needed to thoroughly understand the code, and "why" demonstrates the kind of choices a driver writer needs to make, although scull is definitely not typical as a device.

This section deals only with the memory allocation policy in scull and doesn't show the hardware management skills you need to write real drivers. These skills are introduced in Chapter 9 and Chapter 10. Therefore, you can skip this section if you're not interested in understanding the inner workings of the memory-oriented scull driver.

The region of memory used by scull, also called a device, is variable in length. The more you write, the more it grows; trimming is performed by overwriting the device with a shorter file.

The scull driver introduces two core functions used to manage memory in the Linux kernel. These functions, defined in <*linux/slab.h*>, are:

```
void *kmalloc(size_t size, int flags);

void kfree(void *ptr);
```

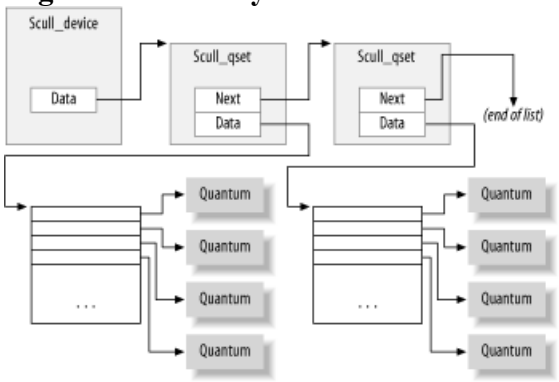
A call to kmalloc attempts to allocate size bytes of memory; the return value is a pointer to that memory or NULL if the allocation fails. The flags argument is used to describe how the memory should be allocated; we examine those flags in detail in Chapter 8. For now, we always use GFP_KERNEL. Allocated memory should be freed with kfree. You should never pass anything to kfree that was not obtained from kmalloc. It is, however, legal to pass a NULL pointer to kfree.

kmalloc is not the most efficient way to allocate large areas of memory (see Chapter 8), so the implementation chosen for scull is not a particularly smart one. The source code for a smart implementation would be more difficult to read, and the aim of this section is to show read and write, not memory management. That's why the code just uses kmalloc and kfree without resorting to allocation of whole pages, although that approach would be more efficient.

On the flip side, we didn't want to limit the size of the "device" area, for both a philosophical reason and a practical one. Philosophically, it's always a bad idea to put arbitrary limits on data items being managed. Practically, scull can be used to temporarily eat up your system's memory in order to run tests under low-memory conditions. Running such tests might help you understand the system's internals. You can use the command cp /dev/zero /dev/scull0 to eat all the real RAM with scull, and you can use the dd utility to choose how much data is copied to the scull device.

In scull, each device is a linked list of pointers, each of which points to a scull_dev structure. Each such structure can refer, by default, to at most four million bytes, through an array of intermediate pointers. The released source uses an array of 1000 pointers to areas of 4000 bytes. We call each memory area a *quantum* and the array (or its length) a *quantum set* . A scull device and its memory areas are shown in Figure 3-1.

Figure 3-1. The layout of a scull device



The chosen numbers are such that writing a single byte in scull consumes 8000 or 12,000 thousand bytes of memory: 4000 for the quantum and 4000 or 8000 for the quantum set (according to whether a pointer is

3.7. read and write

The read and write methods both perform a similar task, that is, copying data from and to application code. Therefore, their prototypes are pretty similar, and it's worth introducing them at the same time:

```
ssize_t read(struct file *filp, char _ _user *buff,
    size_t count, loff_t *offp);
ssize_t write(struct file *filp, const char _ _user *buff,
    size_t count, loff_t *offp);
```

For both methods, filp is the file pointer and count is the size of the requested data transfer. The buff argument points to the user buffer holding the data to be written or the empty buffer where the newly read data should be placed. Finally, offp is a pointer to a "long offset type" object that indicates the file position the user is accessing. The return value is a "signed size type"; its use is discussed later.

Let us repeat that the buff argument to the read and write methods is a user-space pointer. Therefore, it cannot be directly dereferenced by kernel code. There are a few reasons for this restriction:

- Depending on which architecture your driver is running on, and how the kernel was configured, the user-space pointer may not be valid while running in kernel mode at all. There may be no mapping for that address, or it could point to some other, random data.
- Even if the pointer does mean the same thing in kernel space, user-space memory is paged, and the memory in question might not be resident in RAM when the system call is made. Attempting to reference the user-space memory directly could generate a page fault, which is something that kernel code is not allowed to do. The result would be an "oops," which would result in the death of the process that made the system call.
- The pointer in question has been supplied by a user program, which could be buggy or malicious. If your driver ever blindly dereferences a user-supplied pointer, it provides an open doorway allowing a user-space program to access or overwrite memory anywhere in the system. If you do not wish to be responsible for compromising the security of your users' systems, you cannot ever dereference a user-space pointer directly.

Obviously, your driver must be able to access the user-space buffer in order to get its job done. This access must always be performed by special, kernel-supplied functions, however, in order to be safe. We introduce some of those functions (which are defined in *<asm/uaccess.h>*) here, and the rest in the Section 6.1.4; they use some special, architecture-dependent magic to ensure that data transfers between kernel and user space happen in a safe and correct way.

The code for read and write in scull needs to copy a whole segment of data to or from the user address space. This capability is offered by the following kernel functions, which copy an arbitrary array of bytes and sit at the heart of most read and write implementations:

```
unsigned long copy_to_user(void _ _user *to,
                           const void *from,
                           unsigned long count);
unsigned long copy_from_user(void *to,
                             const void _ _user *from,
                             unsigned long count);
```

Although these functions behave like normal memcpy functions, a little extra care must be used when accessing

3.8. Playing with the New Devices

Once you are equipped with the four methods just described, the driver can be compiled and tested; it retains any data you write to it until you overwrite it with new data. The device acts like a data buffer whose length is limited only by the amount of real RAM available. You can try using cp, dd, and input/output redirection to test out the driver.

The free command can be used to see how the amount of free memory shrinks and expands according to how much data is written into scull.

To get more confident with reading and writing one quantum at a time, you can add a printk at an appropriate point in the driver and watch what happens while an application reads or writes large chunks of data. Alternatively, use the strace utility to monitor the system calls issued by a program, together with their return values. Tracing a cp or an ls -l > /dev/scull0 shows quantized reads and writes. Monitoring (and debugging) techniques are presented in detail in Chapter 4

3.9. Quick Reference

This chapter introduced the following symbols and header files. The list of the fields in struct file_operations and struct file is not repeated here.

#include <linux/types.h>

dev_t

dev_t is the type used to r epresent device numbers within the kernel.

int MAJOR(dev_t dev);

int MINOR(dev_t dev);

Macros that extract the major and minor numbers from a device number.

dev_t MKDEV(unsigned int major, unsigned int minor);

Macro that builds a dev_t data item from the major and minor numbers.

#include <linux/fs.h>

The "filesystem" header is the header required for writing device drivers. Many important functions and data structures are declared in here.

int register_chrdev_region(dev_t first, unsigned int count, char *name)

int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int

count, char *name)

void unregister_chrdev_region(dev_t first, unsigned int count);

Functions that allow a driver to allocate and free ranges of device numbers. register_chrdev_region should be used when the desired major number is known in advance; for dynamic allocation, use alloc_chrdev_region instead.

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

*fops);

The old (pre-2.6) char device registration routine. It is emulated in the 2.6 kernel but should not be used for new code. If the major number is not 0, it is used unchanged; otherwise a dynamic number is assigned for this device.

int unregister_chrdev(unsigned int major, const char *name);

Chapter 4. Debugging Techniques

Kernel programming brings its own, unique debugging challenges. Kernel code cannot be easily executed under a debugger, nor can it be easily traced, because it is a set of functionalities not related to a specific process. Kernel code errors can also be exceedingly hard to reproduce and can bring down the entire system with them, thus destroying much of the evidence that could be used to track them down.

This chapter introduces techniques you can use to monitor kernel code and trace errors under such trying circumstances.

4.1. Debugging Support in the Kernel

In Chapter 2, we recommended that you build and install your own kernel, rather than running the stock kernel that comes with your distribution. One of the strongest reasons for running your own kernel is that the kernel developers have built several debugging features into the kernel itself. These features can create extra output and slow performance, so they tend not to be enabled in production kernels from distributors. As a kernel developer, however, you have different priorities and will gladly accept the (minimal) overhead of the extra kernel debugging support.

Here, we list the configuration options that should be enabled for kernels used for development. Except where specified otherwise, all of these options are found under the "kernel hacking" menu in whatever kernel configuration tool you prefer. Note that some of these options are not supported by all architectures.

CONFIG_DEBUG_KERNEL

This option just makes other debugging options available; it should be turned on but does not, by itself, enable any features.

CONFIG_DEBUG_SLAB

This crucial option turns on several types of checks in the kernel memory allocation functions; with these checks enabled, it is possible to detect a number of memory overrun and missing initialization errors. Each byte of allocated memory is set to 0xa5 before being handed to the caller and then set to 0x6b when it is freed. If you ever see either of those "poison" patterns repeating in output from your driver (or often in an oops listing), you'll know exactly what sort of error to look for. When debugging is enabled, the kernel also places special guard values before and after every allocated memory object; if those values ever get changed, the kernel knows that somebody has overrun a memory allocation, and it complains loudly. Various checks for more obscure errors are enabled as well.

CONFIG_DEBUG_PAGEALLOC

Full pages are removed from the kernel address space when freed. This option can slow things down significantly, but it can also quickly point out certain kinds of memory corruption errors.

CONFIG_DEBUG_SPINLOCK

With this option enabled, the kernel catches operations on uninitialized spinlocks and various other errors (such as unlocking a lock twice).

CONFIG_DEBUG_SPINLOCK_SLEEP

This option enables a check for attempts to sleep while holding a spinlock. In fact, it complains if you call a function that could potentially sleep, even if the call in question would not sleep.

CONFIG_INIT_DEBUG

Items marked with __init (or __initdata) are discarded after system initialization or module load time. This option enables checks for code that attempts to access initialization-time memory after initialization is complete.

CONFIG_DEBUG_INFO

This option causes the kernel to be built with full debugging information included. You'll need that information if you want to debug the kernel with gdb. You may also want to enable CONFIG_FRAME_POINTER if you

4.2. Debugging by Printing

The most common debugging technique is monitoring, which in applications programming is done by calling printf at suitable points. When you are debugging kernel code, you can accomplish the same goal with printk.

4.2.1. printk

We used the printk function in earlier chapters with the simplifying assumption that it works like printf. Now it's time to introduce some of the differences.

One of the differences is that printk lets you classify messages according to their severity by associating different loglevels, or priorities, with the messages. You usually indicate the loglevel with a macro. For example, KERN_INFO, which we saw prepended to some of the earlier print statements, is one of the possible loglevels of the message. The loglevel macro expands to a string, which is concatenated to the message text at compile time; that's why there is no comma between the priority and the format string in the following examples. Here are two examples of printk commands, a debug message and a critical message:

```
printk(KERN_DEBUG "Here I am: %s:%i\n", _ _FILE_ _, _ _LINE_ _);

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

There are eight possible loglevel strings, defined in the header *<linux/kernel.h>*; we list them in order of decreasing severity:

KERN_EMERG

Used for emergency messages, usually those that precede a crash.

KERN_ALERT

A situation requiring immediate action.

KERN_CRIT

Critical conditions, often related to serious hardware or software failures.

KERN_ERR

Used to report error conditions; device drivers often use KERN_ERR to report hardware difficulties.

KERN_WARNING

Warnings about problematic situations that do not, in themselves, create serious problems with the system.

KERN_NOTICE

Situations that are normal, but still worthy of note. A number of security-related conditions are reported at this level.

KERN_INFO

Informational messages. Many drivers print information about the hardware they find at startup time at this level.

4.3. Debugging by Querying

The previous section described how printk works and how it can be used. What it didn't talk about are its disadvantages.

A massive use of printk can slow down the system noticeably, even if you lower console_loglevel to avoid loading the console device, because syslogd keeps syncing its output files; thus, every line that is printed causes a disk operation. This is the right implementation from syslogd 's perspective. It tries to write everything to disk in case the system crashes right after printing the message; however, you don't want to slow down your system just for the sake of debugging messages. This problem can be solved by prefixing the name of your log file as it appears in */etc/syslogd.conf* with a hyphen.[2] The problem with changing the configuration file is that the modification will likely remain there after you are done debugging, even though during normal system operation you do want messages to be flushed to disk as soon as possible. An alternative to such a permanent change is running a program other than klogd (such as cat /proc/kmsg, as suggested earlier), but this may not provide a suitable environment for normal system operation.

[2] The hyphen, or minus sign, is a "magic" marker to prevent syslogd from flushing the file to disk at every new message, documented in syslog.conf(5), a manpage worth reading.

More often than not, the best way to get relevant information is to query the system when you need the information, instead of continually producing data. In fact, every Unix system provides many tools for obtaining system information: ps, netstat, vmstat, and so on.

A few techniques are available to driver developers for querying the system: creating a file in the */proc* filesystem, using the ioctl driver method, and exporting attributes via sysfs. The use of sysfs requires quite some background on the driver model. It is discussed in Chapter 14.

4.3.1. Using the /proc Filesystem

The */proc* filesystem is a special, software-created filesystem that is used by the kernel to export information to the world. Each file under */proc* is tied to a kernel function that generates the file's "contents" on the fly when the file is read. We have already seen some of these files in action; */proc/modules*, for example, always returns a list of the currently loaded modules.

/proc is heavily used in the Linux system. Many utilities on a modern Linux distribution, such as ps, top, and uptime, get their information from */proc*. Some device drivers also export information via */proc*, and yours can do so as well. The */proc* filesystem is dynamic, so your module can add or remove entries at any time.

Fully featured */proc* entries can be complicated beasts; among other things, they can be written to as well as read from. Most of the time, however, */proc* entries are read-only files. This section concerns itself with the simple read-only case. Those who are interested in implementing something more complicated can look here for the basics; the kernel source may then be consulted for the full picture.

Before we continue, however, we should mention that adding files under */proc* is discouraged. The */proc* filesystem is seen by the kernel developers as a bit of an uncontrolled mess that has gone far beyond its original purpose (which was to provide information about the processes running in the system). The recommended way of making information available in new code is via sysfs. As suggested, working with sysfs requires an understanding of the Linux device model, however, and we do not get to that until Chapter 14. Meanwhile, files under */proc* are slightly easier to create, and they are entirely suitable for debugging purposes, so we cover them here.

4.3.1.1 Implementing files in /proc

All modules that work with */proc* should include *<linux/proc_fs.h>* to define the proper functions.

To create a read-only */proc* file, your driver must implement a function to produce the data when the file is read. When some process reads the file (using the read system call), the request reaches your module by means of this function. We'll look at this function first and get to the registration interface later in this section.

When a process reads from your */proc* file, the kernel allocates a page of memory (i.e., PAGE_SIZE bytes)

4.4. Debugging by Watching

Sometimes minor problems can be tracked down by watching the behavior of an application in user space. Watching programs can also help in building confidence that a driver is working correctly. For example, we were able to feel confident about scull after looking at how its read implementation reacted to read requests for different amounts of data.

There are various ways to watch a user-space program working. You can run a debugger on it to step through its functions, add print statements, or run the program under strace. Here we'll discuss just the last technique, which is most interesting when the real goal is examining kernel code.

The strace command is a powerful tool that shows all the system calls issued by a user-space program. Not only does it show the calls, but it can also show the arguments to the calls and their return values in symbolic form. When a system call fails, both the symbolic value of the error (e.g., ENOMEM) and the corresponding string (Out of memory) are displayed. strace has many command-line options; the most useful of which are -t to display the time when each call is executed, -T to display the time spent in the call, -e to limit the types of calls traced, and -o to redirect the output to a file. By default, strace prints tracing information on stderr.

strace receives information from the kernel itself. This means that a program can be traced regardless of whether or not it was compiled with debugging support (the -g option to gcc) and whether or not it is stripped. You can also attach tracing to a running process, similar to the way a debugger can connect to a running process and control it.

The trace information is often used to support bug reports sent to application developers, but it's also invaluable to kernel programmers. We've seen how driver code executes by reacting to system calls; strace allows us to check the consistency of input and output data of each call.

For example, the following screen dump shows (most of) the last lines of running the command strace ls /dev > /dev/scull0 :

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

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

fcntl64(3, F_SETFD, FD_CLOEXEC)         = 0

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

[...]

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

close(3)                                = 0

[...]

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

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

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

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

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

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

close(1)                                = 0

exit_group(0)                           = ?
```

It's apparent from the first write call that after ls finished looking in the target directory, it tried to write 4 KB. Strangely (for ls), only 4000 bytes were written, and the operation was retried. However, we know that the

4.5. Debugging System Faults

Even if you've used all the monitoring and debugging techniques, sometimes bugs remain in the driver, and the system faults when the driver is executed. When this happens, it's important to be able to collect as much information as possible to solve the problem.

Note that "fault" doesn't mean "panic." The Linux code is robust enough to respond gracefully to most errors: a fault usually results in the destruction of the current process while the system goes on working. The system can panic, and it may if a fault happens outside of a process's context or if some vital part of the system is compromised. But when the problem is due to a driver error, it usually results only in the sudden death of the process unlucky enough to be using the driver. The only unrecoverable damage when a process is destroyed is that some memory allocated to the process's context is lost; for instance, dynamic lists allocated by the driver through kmalloc might be lost. However, since the kernel calls the close operation for any open device when a process dies, your driver can release what was allocated by the open method.

Even though an oops usually does not bring down the entire system, you may well find yourself needing to reboot after one happens. A buggy driver can leave hardware in an unusable state, leave kernel resources in an inconsistent state, or, in the worst case, corrupt kernel memory in random places. Often you can simply unload your buggy driver and try again after an oops. If, however, you see anything that suggests that the system as a whole is not well, your best bet is usually to reboot immediately.

We've already said that when kernel code misbehaves, an informative message is printed on the console. The next section explains how to decode and use such messages. Even though they appear rather obscure to the novice, processor dumps are full of interesting information, often sufficient to pinpoint a program bug without the need for additional testing.

4.5.1. Oops Messages

Most bugs show themselves in NULL pointer dereferences or by the use of other incorrect pointer values. The usual outcome of such bugs is an oops message.

Almost any address used by the processor is a virtual address and is mapped to physical addresses through a complex structure of page tables (the exceptions are physical addresses used with the memory management subsystem itself). When an invalid pointer is dereferenced, the paging mechanism fails to map the pointer to a physical address, and the processor signals a page fault to the operating system. If the address is not valid, the kernel is not able to "page in" the missing address; it (usually) generates an oops if this happens while the processor is in supervisor mode.

An oops displays the processor status at the time of the fault, including the contents of the CPU registers and other seemingly incomprehensible information. The message is generated by printk statements in the fault handler (*arch/*/kernel/traps.c*) and is dispatched as described earlier in Section 4.2.1).

Let's look at one such message. Here's what results from dereferencing a NULL pointer on a PC running Version 2.6 of the kernel. The most relevant information here is the instruction pointer (EIP), the address of the faulty instruction.

```
 Unable to handle kernel NULL pointer dereference at virtual address 00000000

 printing eip:

d083a064

Oops: 0002 [#1]

SMP

CPU:    0

EIP:    0060:[<d083a064>]    Not tainted

EFLAGS: 00010246   (2.6.6)

EIP is at faulty_write+0x4/0x10 [faulty]
```

4.6. Debuggers and Related Tools

The last resort in debugging modules is using a debugger to step through the code, watching the value of variables and machine registers. This approach is time-consuming and should be avoided whenever possible. Nonetheless, the fine-grained perspective on the code that is achieved through a debugger is sometimes invaluable.

Using an interactive debugger on the kernel is a challenge. The kernel runs in its own address space on behalf of all the processes on the system. As a result, a number of common capabilities provided by user-space debuggers, such as breakpoints and single-stepping, are harder to come by in the kernel. In this section we look at several ways of debugging the kernel; each of them has advantages and disadvantages.

4.6.1. Using gdb

gdb can be quite useful for looking at the system internals. Proficient use of the debugger at this level requires some confidence with gdb commands, some understanding of assembly code for the target platform, and the ability to match source code and optimized assembly.

The debugger must be invoked as though the kernel were an application. In addition to specifying the filename for the ELF kernel image, you need to provide the name of a core file on the command line. For a running kernel, that core file is the kernel core image, */proc/kcore*. A typical invocation of gdb looks like the following:

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

The first argument is the name of the uncompressed ELF kernel executable, not the *zImage* or *bzImage* or anything built specifically for the boot environment.

The second argument on the gdb command line is the name of the core file. Like any file in */proc*, */proc/kcore* is generated when it is read. When the read system call executes in the */proc* filesystem, it maps to a data-generation function rather than a data-retrieval one; we've already exploited this feature in the section Section 4.3.1. *kcore* is used to represent the kernel "executable" in the format of a core file; it is a huge file, because it represents the whole kernel address space, which corresponds to all physical memory. From within gdb, you can look at kernel variables by issuing the standard gdb commands. For example, p jiffies prints the number of clock ticks from system boot to the current time.

When you print data from gdb, the kernel is still running, and the various data items have different values at different times; gdb, however, optimizes access to the core file by caching data that has already been read. If you try to look at the jiffies variable once again, you'll get the same answer as before. Caching values to avoid extra disk access is a correct behavior for conventional core files but is inconvenient when a "dynamic" core image is used. The solution is to issue the command core-file /proc/kcore whenever you want to flush the gdb cache; the debugger gets ready to use a new core file and discards any old information. You won't, however, always need to issue core-file when reading a new datum; gdb reads the core in chunks of a few kilobytes and caches only chunks it has already referenced.

Numerous capabilities normally provided by gdb are not available when you are working with the kernel. For example, gdb is not able to modify kernel data; it expects to be running a program to be debugged under its own control before playing with its memory image. It is also not possible to set breakpoints or watchpoints, or to single-step through kernel functions.

Note that, in order to have symbol information available for gdb, you must compile your kernel with the CONFIG_DEBUG_INFO option set. The result is a far larger kernel image on disk, but, without that information, digging through kernel variables is almost impossible.

With the debugging information available, you can learn a lot about what is going on inside the kernel. gdb happily prints out structures, follows pointers, etc. One thing that is harder, however, is examining modules. Since modules are not part of the *vmlinux* image passed to gdb, the debugger knows nothing about them. Fortunately, as of kernel 2.6.7, it is possible to teach gdb what it needs to know to examine loadable modules.

Linux loadable modules are ELF-format executable images; as such, they have been divided up into numerous sections. A typical module can contain a dozen or more sections, but there are typically three that are relevant in a debugging session:

Chapter 5. Concurrency and Race Conditions

Thus far, we have paid little attention to the problem of concurrency—i.e., what happens when the system tries to do more than one thing at once. The management of concurrency is, however, one of the core problems in operating systems programming. Concurrency-related bugs are some of the easiest to create and some of the hardest to find. Even expert Linux kernel programmers end up creating concurrency-related bugs on occasion.

In early Linux kernels, there were relatively few sources of concurrency. Symmetric multiprocessing (SMP) systems were not supported by the kernel, and the only cause of concurrent execution was the servicing of hardware interrupts. That approach offers simplicity, but it no longer works in a world that prizes performance on systems with more and more processors, and that insists that the system respond to events quickly. In response to the demands of modern hardware and applications, the Linux kernel has evolved to a point where many more things are going on simultaneously. This evolution has resulted in far greater performance and scalability. It has also, however, significantly complicated the task of kernel programming. Device driver programmers must now factor concurrency into their designs from the beginning, and they must have a strong understanding of the facilities provided by the kernel for concurrency management.

The purpose of this chapter is to begin the process of creating that understanding. To that end, we introduce facilities that are immediately applied to the scull driver from Chapter 3. Other facilities presented here are not put to use for some time yet. But first, we take a look at what could go wrong with our simple scull driver and how to avoid these potential problems.

5.1. Pitfalls in scull

Let us take a quick look at a fragment of the scull memory management code. Deep down inside the write logic, scull must decide whether the memory it requires has been allocated yet or not. One piece of the code that handles this task is:

```
    if (!dptr->data[s_pos]) {

        dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);

        if (!dptr->data[s_pos])

            goto out;

    }
```

Suppose for a moment that two processes (we'll call them "A" and "B") are independently attempting to write to the same offset within the same scull device. Each process reaches the if test in the first line of the fragment above at the same time. If the pointer in question is NULL, each process will decide to allocate memory, and each will assign the resulting pointer to dptr->data[s_pos]. Since both processes are assigning to the same location, clearly only one of the assignments will prevail.

What will happen, of course, is that the process that completes the assignment second will "win." If process A assigns first, its assignment will be overwritten by process B. At that point, scull will forget entirely about the memory that A allocated; it only has a pointer to B's memory. The memory allocated by A, thus, will be dropped and never returned to the system.

This sequence of events is a demonstration of a *race condition* . Race conditions are a result of uncontrolled access to shared data. When the wrong access pattern happens, something unexpected results. For the race condition discussed here, the result is a memory leak. That is bad enough, but race conditions can often lead to system crashes, corrupted data, or security problems as well. Programmers can be tempted to disregard race conditions as extremely low probability events. But, in the computing world, one-in-a-million events can happen every few seconds, and the consequences can be grave.

We will eliminate race conditions from scull shortly, but first we need to take a more general view of concurrency.

5.2. Concurrency and Its Management

In a modern Linux system, there are numerous sources of concurrency and, therefore, possible race conditions. Multiple user-space processes are running, and they can access your code in surprising combinations of ways. SMP systems can be executing your code simultaneously on different processors. Kernel code is preemptible; your driver's code can lose the processor at any time, and the process that replaces it could also be running in your driver. Device interrupts are asynchronous events that can cause concurrent execution of your code. The kernel also provides various mechanisms for delayed code execution, such as workqueues, tasklets, and timers, which can cause your code to run at any time in ways unrelated to what the current process is doing. In the modern, hot-pluggable world, your device could simply disappear while you are in the middle of working with it.

Avoidance of race conditions can be an intimidating task. In a world where anything can happen at any time, how does a driver programmer avoid the creation of absolute chaos? As it turns out, most race conditions can be avoided through some thought, the kernel's concurrency control primitives, and the application of a few basic principles. We'll start with the principles first, then get into the specifics of how to apply them.

Race conditions come about as a result of shared access to resources. When two threads of execution[1] have a reason to work with the same data structures (or hardware resources), the potential for mixups always exists. So the first rule of thumb to keep in mind as you design your driver is to avoid shared resources whenever possible. If there is no concurrent access, there can be no race conditions. So carefully-written kernel code should have a minimum of sharing. The most obvious application of this idea is to avoid the use of global variables. If you put a resource in a place where more than one thread of execution can find it, there should be a strong reason for doing so.

[1] For the purposes of this chapter, a "thread" of execution is any context that is running code. Each process is clearly a thread of execution, but so is an interrupt handler or other code running in response to an asynchronous kernel event.

The fact of the matter is, however, that such sharing is often required. Hardware resources are, by their nature, shared, and software resources also must often be available to more than one thread. Bear in mind as well that global variables are far from the only way to share data; any time your code passes a pointer to some other part of the kernel, it is potentially creating a new sharing situation. Sharing is a fact of life.

Here is the hard rule of resource sharing: any time that a hardware or software resource is shared beyond a single thread of execution, and the possibility exists that one thread could encounter an inconsistent view of that resource, you must explicitly manage access to that resource. In the scull example above, process B's view of the situation is inconsistent; unaware that process A has already allocated memory for the (shared) device, it performs its own allocation and overwrites A's work. In this case, we must control access to the scull data structure. We need to arrange things so that the code either sees memory that has been allocated or knows that no memory has been or will be allocated by anybody else. The usual technique for access management is called *locking* or *mutual exclusion*—making sure that only one thread of execution can manipulate a shared resource at any time. Much of the rest of this chapter will be devoted to locking.

First, however, we must briefly consider one other important rule. When kernel code creates an object that will be shared with any other part of the kernel, that object must continue to exist (and function properly) until it is known that no outside references to it exist. The instant that scull makes its devices available, it must be prepared to handle requests on those devices. And scull must continue to be able to handle requests on its devices until it knows that no reference (such as open user-space files) to those devices exists. Two requirements come out of this rule: no object can be made available to the kernel until it is in a state where it can function properly, and references to such objects must be tracked. In most cases, you'll find that the kernel handles reference counting for you, but there are always exceptions.

Following the above rules requires planning and careful attention to detail. It is easy to be surprised by concurrent access to resources you hadn't realized were shared. With some effort, however, most race conditions can be headed off before they bite you—or your users.

5.3. Semaphores and Mutexes

So let us look at how we can add locking to scull. Our goal is to make our operations on the scull data structure *atomic*, meaning that the entire operation happens at once as far as other threads of execution are concerned. For our memory leak example, we need to ensure that if one thread finds that a particular chunk of memory must be allocated, it has the opportunity to perform that allocation before any other thread can make that test. To this end, we must set up *critical sections*: code that can be executed by only one thread at any given time.

Not all critical sections are the same, so the kernel provides different primitives for different needs. In this case, every access to the scull data structure happens in process context as a result of a direct user request; no accesses will be made from interrupt handlers or other asynchronous contexts. There are no particular latency (response time) requirements; application programmers understand that I/O requests are not usually satisfied immediately. Furthermore, the scull is not holding any other critical system resource while it is accessing its own data structures. What all this means is that if the scull driver goes to sleep while waiting for its turn to access the data structure, nobody is going to mind.

"Go to sleep" is a well-defined term in this context. When a Linux process reaches a point where it cannot make any further processes, it goes to sleep (or "blocks"), yielding the processor to somebody else until some future time when it can get work done again. Processes often sleep when waiting for I/O to complete. As we get deeper into the kernel, we will encounter a number of situations where we cannot sleep. The write method in scull is not one of those situations, however. So we can use a locking mechanism that might cause the process to sleep while waiting for access to the critical section.

Just as importantly, we will be performing an operation (memory allocation with kmalloc) that could sleep—so sleeps are a possibility in any case. If our critical sections are to work properly, we must use a locking primitive that works when a thread that owns the lock sleeps. Not all locking mechanisms can be used where sleeping is a possibility (we'll see some that don't later in this chapter). For our present needs, however, the mechanism that fits best is a *semaphore*.

Semaphores are a well-understood concept in computer science. At its core, a semaphore is a single integer value combined with a pair of functions that are typically called P and V. A process wishing to enter a critical section will call P on the relevant semaphore; if the semaphore's value is greater than zero, that value is decremented by one and the process continues. If, instead, the semaphore's value is 0 (or less), the process must wait until somebody else releases the semaphore. Unlocking a semaphore is accomplished by calling V; this function increments the value of the semaphore and, if necessary, wakes up processes that are waiting.

When semaphores are used for *mutual exclusion*—keeping multiple processes from running within a critical section simultaneously—their value will be initially set to 1. Such a semaphore can be held only by a single process or thread at any given time. A semaphore used in this mode is sometimes called a *mutex*, which is, of course, an abbreviation for "mutual exclusion." Almost all semaphores found in the Linux kernel are used for mutual exclusion.

5.3.1. The Linux Semaphore Implementation

The Linux kernel provides an implementation of semaphores that conforms to the above semantics, although the terminology is a little different. To use semaphores, kernel code must include *<asm/semaphore.h>*. The relevant type is struct semaphore; actual semaphores can be declared and initialized in a few ways. One is to create a semaphore directly, then set it up with sema_init:

```
void sema_init(struct semaphore *sem, int val);
```

where val is the initial value to assign to a semaphore.

Usually, however, semaphores are used in a mutex mode. To make this common case a little easier, the kernel has provided a set of helper functions and macros. Thus, a mutex can be declared and initialized with one of the following:

```
DECLARE_MUTEX(name);

DECLARE_MUTEX_LOCKED(name);
```

Here, the result is a semaphore variable (called name) that is initialized to 1 (with DECLARE_MUTEX) or 0

5.4. Completions

A common pattern in kernel programming involves initiating some activity outside of the current thread, then waiting for that activity to complete. This activity can be the creation of a new kernel thread or user-space process, a request to an existing process, or some sort of hardware-based action. It such cases, it can be tempting to use a semaphore for synchronization of the two tasks, with code such as:

```
struct semaphore sem;



init_MUTEX_LOCKED(&sem);

start_external_task(&sem);

down(&sem);
```

The external task can then call up(&sem) when its work is done.

As is turns out, semaphores are not the best tool to use in this situation. In normal use, code attempting to lock a semaphore finds that semaphore available almost all the time; if there is significant contention for the semaphore, performance suffers and the locking scheme needs to be reviewed. So semaphores have been heavily optimized for the "available" case. When used to communicate task completion in the way shown above, however, the thread calling down will almost always have to wait; performance will suffer accordingly. Semaphores can also be subject to a (difficult) race condition when used in this way if they are declared as automatic variables. In some cases, the semaphore could vanish before the process calling up is finished with it.

These concerns inspired the addition of the "completion" interface in the 2.4.7 kernel. Completions are a lightweight mechanism with one task: allowing one thread to tell another that the job is done. To use completions, your code must include *<linux/completion.h>*. A completion can be created with:

```
DECLARE_COMPLETION(my_completion);
```

Or, if the completion must be created and initialized dynamically:

```
struct completion my_completion;

/* ... */

init_completion(&my_completion);
```

Waiting for the completion is a simple matter of calling:

```
void wait_for_completion(struct completion *c);
```

Note that this function performs an uninterruptible wait. If your code calls wait_for_completion and nobody ever completes the task, the result will be an unkillable process.[2]

[2] As of this writing, patches adding interruptible versions were in circulation but had not been merged into the mainline.

On the other side, the actual completion event may be signalled by calling one of the following:

```
void complete(struct completion *c);

void complete_all(struct completion *c);
```

The two functions behave differently if more than one thread is waiting for the same completion event. complete wakes up only one of the waiting threads while complete_all allows all of them to proceed. In most cases, there is only one waiter, and the two functions will produce an identical result.

A completion is normally a one-shot device; it is used once then discarded. It is possible, however, to reuse completion structures if proper care is taken. If complete_all is not used, a completion structure can be reused without any problems as long as there is no ambiguity about what event is being signalled. If you use complete_all, however, you must reinitialize the completion structure before reusing it. The macro:

```
INIT_COMPLETION(struct completion c);
```

5.5. Spinlocks

Semaphores are a useful tool for mutual exclusion, but they are not the only such tool provided by the kernel. Instead, most locking is implemented with a mechanism called a *spinlock*. Unlike semaphores, spinlocks may be used in code that cannot sleep, such as interrupt handlers. When properly used, spinlocks offer higher performance than semaphores in general. They do, however, bring a different set of constraints on their use.

Spinlocks are simple in concept. A spinlock is a mutual exclusion device that can have only two values: "locked" and "unlocked." It is usually implemented as a single bit in an integer value. Code wishing to take out a particular lock tests the relevant bit. If the lock is available, the "locked" bit is set and the code continues into the critical section. If, instead, the lock has been taken by somebody else, the code goes into a tight loop where it repeatedly checks the lock until it becomes available. This loop is the "spin" part of a spinlock.

Of course, the real implementation of a spinlock is a bit more complex than the description above. The "test and set" operation must be done in an atomic manner so that only one thread can obtain the lock, even if several are spinning at any given time. Care must also be taken to avoid deadlocks on *hyperthreaded* processors—chips that implement multiple, virtual CPUs sharing a single processor core and cache. So the actual spinlock implementation is different for every architecture that Linux supports. The core concept is the same on all systems, however, when there is contention for a spinlock, the processors that are waiting execute a tight loop and accomplish no useful work.

Spinlocks are, by their nature, intended for use on multiprocessor systems, although a uniprocessor workstation running a preemptive kernel behaves like SMP, as far as concurrency is concerned. If a nonpreemptive uniprocessor system ever went into a spin on a lock, it would spin forever; no other thread would ever be able to obtain the CPU to release the lock. For this reason, spinlock operations on uniprocessor systems without preemption enabled are optimized to do nothing, with the exception of the ones that change the IRQ masking status. Because of preemption, even if you never expect your code to run on an SMP system, you still need to implement proper locking.

5.5.1. Introduction to the Spinlock API

The required include file for the spinlock primitives is *<linux/spinlock.h>*. An actual lock has the type spinlock_t. Like any other data structure, a spinlock must be initialized. This initialization may be done at compile time as follows:

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
```

or at runtime with:

```
void spin_lock_init(spinlock_t *lock);
```

Before entering a critical section, your code must obtain the requisite lock with:

```
void spin_lock(spinlock_t *lock);
```

Note that all spinlock waits are, by their nature, uninterruptible. Once you call spin_lock, you will spin until the lock becomes available.

To release a lock that you have obtained, pass it to:

```
void spin_unlock(spinlock_t *lock);
```

There are many other spinlock functions, and we will look at them all shortly. But none of them depart from the core idea shown by the functions listed above. There is very little that one can do with a lock, other than lock and release it. However, there are a few rules about how you must work with spinlocks. We will take a moment to look at those before getting into the full spinlock interface.

5.5.2. Spinlocks and Atomic Context

Imagine for a moment that your driver acquires a spinlock and goes about its business within its critical section. Somewhere in the middle, your driver loses the processor. Perhaps it has called a function (copy_from_user, say) that puts the process to sleep. Or, perhaps, kernel preemption kicks in, and a higher-priority process pushes your code aside. Your code is now holding a lock that it will not release any time in the foreseeable future. If some other thread tries to obtain the same lock, it will, in the best case, wait (spinning in the processor) for a

5.6. Locking Traps

Many years of experience with locks—experience that predates Linux—have shown that locking can be very hard to get right. Managing concurrency is an inherently tricky undertaking, and there are many ways of making mistakes. In this section, we take a quick look at things that can go wrong.

5.6.1. Ambiguous Rules

As has already been said above, a proper locking scheme requires clear and explicit rules. When you create a resource that can be accessed concurrently, you should define which lock will control that access. Locking should really be laid out at the beginning; it can be a hard thing to retrofit in afterward. Time taken at the outset usually is paid back generously at debugging time.

As you write your code, you will doubtless encounter several functions that all require access to structures protected by a specific lock. At this point, you must be careful: if one function acquires a lock and then calls another function that also attempts to acquire the lock, your code deadlocks. Neither semaphores nor spinlocks allow a lock holder to acquire the lock a second time; should you attempt to do so, things simply hang.

To make your locking work properly, you have to write some functions with the assumption that their caller has already acquired the relevant lock(s). Usually, only your internal, static functions can be written in this way; functions called from outside must handle locking explicitly. When you write internal functions that make assumptions about locking, do yourself (and anybody else who works with your code) a favor and document those assumptions explicitly. It can be very hard to come back months later and figure out whether you need to hold a lock to call a particular function or not.

In the case of scull, the design decision taken was to require all functions invoked directly from system calls to acquire the semaphore applying to the device structure that is accessed. All internal functions, which are only called from other scull functions, can then assume that the semaphore has been properly acquired.

5.6.2. Lock Ordering Rules

In systems with a large number of locks (and the kernel is becoming such a system), it is not unusual for code to need to hold more than one lock at once. If some sort of computation must be performed using two different resources, each of which has its own lock, there is often no alternative to acquiring both locks.

Taking multiple locks can be dangerous, however. If you have two locks, called Lock1 and Lock2, and code needs to acquire both at the same time, you have a potential deadlock. Just imagine one thread locking Lock1 while another simultaneously takes Lock2. Then each thread tries to get the one it doesn't have. Both threads will deadlock.

The solution to this problem is usually simple: when multiple locks must be acquired, they should always be acquired in the same order. As long as this convention is followed, simple deadlocks like the one described above can be avoided. However, following lock ordering rules can be easier said than done. It is very rare that such rules are actually written down anywhere. Often the best you can do is to see what other code does.

A couple of rules of thumb can help. If you must obtain a lock that is local to your code (a device lock, say) along with a lock belonging to a more central part of the kernel, take your lock first. If you have a combination of semaphores and spinlocks, you must, of course, obtain the semaphore(s) first; calling down (which can sleep) while holding a spinlock is a serious error. But most of all, try to avoid situations where you need more than one lock.

5.6.3. Fine- Versus Coarse-Grained Locking

The first Linux kernel that supported multiprocessor systems was 2.0; it contained exactly one spinlock. The *big kernel lock* turned the entire kernel into one large critical section; only one CPU could be executing kernel code at any given time. This lock solved the concurrency problem well enough to allow the kernel developers to address all of the other issues involved in supporting SMP. But it did not scale very well. Even a two-processor system could spend a significant amount of time simply waiting for the big kernel lock. The performance of a four-processor system was not even close to that of four independent machines.

5.7. Alternatives to Locking

The Linux kernel provides a number of powerful locking primitives that can be used to keep the kernel from tripping over its own feet. But, as we have seen, the design and implementation of a locking scheme is not without its pitfalls. Often there is no alternative to semaphores and spinlocks; they may be the only way to get the job done properly. There are situations, however, where atomic access can be set up without the need for full locking. This section looks at other ways of doing things.

5.7.1. Lock-Free Algorithms

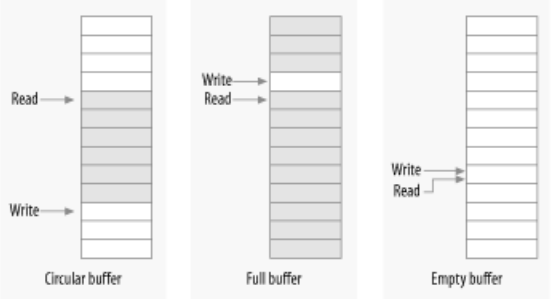
Sometimes, you can recast your algorithms to avoid the need for locking altogether. A number of reader/writer situations—if there is only one writer—can often work in this manner. If the writer takes care that the view of the data structure, as seen by the reader, is always consistent, it may be possible to create a lock-free data structure.

A data structure that can often be useful for lockless producer/consumer tasks is the *circular buffer* . This algorithm involves a producer placing data into one end of an array, while the consumer removes data from the other. When the end of the array is reached, the producer wraps back around to the beginning. So a circular buffer requires an array and two index values to track where the next new value goes and which value should be removed from the buffer next.

When carefully implemented, a circular buffer requires no locking in the absence of multiple producers or consumers. The producer is the only thread that is allowed to modify the write index and the array location it points to. As long as the writer stores a new value into the buffer before updating the write index, the reader will always see a consistent view. The reader, in turn, is the only thread that can access the read index and the value it points to. With a bit of care to ensure that the two pointers do not overrun each other, the producer and the consumer can access the buffer concurrently with no race conditions.

Figure 5-1 shows circular buffer in several states of fill. This buffer has been defined such that an empty condition is indicated by the read and write pointers being equal, while a full condition happens whenever the write pointer is immediately behind the read pointer (being careful to account for a wrap!). When carefully programmed, this buffer can be used without locks.

Figure 5-1. A circular buffer



Circular buffers show up reasonably often in device drivers. Networking adaptors, in particular, often use circular buffers to exchange data (packets) with the processor. Note that, as of 2.6.10, there is a generic circular buffer implementation available in the kernel; see <*linux/kfifo.h*> for information on how to use it.

5.7.2. Atomic Variables

Sometimes, a shared resource is a simple integer value. Suppose your driver maintains a shared variable n_op that tells how many device operations are currently outstanding. Normally, even a simple operation such as:

```
n_op++;
```

would require locking. Some processors might perform that sort of increment in an atomic manner, but you can't count on it. But a full locking regime seems like overhead for a simple integer value. For cases like this, the kernel provides an atomic integer type called atomic_t, defined in <*asm/atomic.h*>.

5.8. Quick Reference

This chapter has introduced a substantial set of symbols for the management of concurrency. The most important of these are summarized here:

#include <asm/semaphore.h>

The include file that defines semaphores and the operations on them.

DECLARE_MUTEX(name);

DECLARE_MUTEX_LOCKED(name);

Two macros for declaring and initializing a semaphore used in mutual exclusion mode.

void init_MUTEX(struct semaphore *sem);

void init_MUTEX_LOCKED(struct semaphore *sem);

These two functions can be used to initialize a semaphore at runtime.

void down(struct semaphore *sem);

int down_interruptible(struct semaphore *sem);

int down_trylock(struct semaphore *sem);

void up(struct semaphore *sem);

Lock and unlock a semaphore. down puts the calling process into an uninterruptible sleep if need be; down_interruptible, instead, can be interrupted by a signal. down_trylock does not sleep; instead, it returns immediately if the semaphore is unavailable. Code that locks a semaphore must eventually unlock it with up.

struct rw_semaphore;

init_rwsem(struct rw_semaphore *sem);

The reader/writer version of semaphores and the function that initializes it.

void down_read(struct rw_semaphore *sem);

int down_read_trylock(struct rw_semaphore *sem);

void up_read(struct rw_semaphore *sem);

Functions for obtaining and releasing read access to a reader/writer semaphore.

Chapter 6. Advanced Char Driver Operations

In Chapter 3, we built a complete device driver that the user can write to and read from. But a real device usually offers more functionality than synchronous read and write. Now that we're equipped with debugging tools should something go awry—and a firm understanding of concurrency issues to help keep things from going awry—we can safely go ahead and create a more advanced driver.

This chapter examines a few concepts that you need to understand to write fully featured char device drivers. We start with implementing the ioctl system call, which is a common interface used for device control. Then we proceed to various ways of synchronizing with user space; by the end of this chapter you have a good idea of how to put processes to sleep (and wake them up), implement nonblocking I/O, and inform user space when your devices are available for reading or writing. We finish with a look at how to implement a few different device access policies within drivers.

The ideas discussed here are demonstrated by way of a couple of modified versions of the scull driver. Once again, everything is implemented using in-memory virtual devices, so you can try out the code yourself without needing to have any particular hardware. By now, you may be wanting to get your hands dirty with real hardware, but that will have to wait until Chapter 9.

6.1. ioctl

Most drivers need—in addition to the ability to read and write the device—the ability to perform various types of hardware control via the device driver. Most devices can perform operations beyond simple data transfers; user space must often be able to request, for example, that the device lock its door, eject its media, report error information, change a baud rate, or self destruct. These operations are usually supported via the ioctl method, which implements the system call by the same name.

In user space, the ioctl system call has the following prototype:

```
int ioctl(int fd, unsigned long cmd, ...);
```

The prototype stands out in the list of Unix system calls because of the dots, which usually mark the function as having a variable number of arguments. In a real system, however, a system call can't actually have a variable number of arguments. System calls must have a well-defined prototype, because user programs can access them only through hardware "gates." Therefore, the dots in the prototype represent not a variable number of arguments but a single optional argument, traditionally identified as char *argp. The dots are simply there to prevent type checking during compilation. The actual nature of the third argument depends on the specific control command being issued (the second argument). Some commands take no arguments, some take an integer value, and some take a pointer to other data. Using a pointer is the way to pass arbitrary data to the ioctl call; the device is then able to exchange any amount of data with user space.

The unstructured nature of the ioctl call has caused it to fall out of favor among kernel developers. Each ioctl command is, essentially, a separate, usually undocumented system call, and there is no way to audit these calls in any sort of comprehensive manner. It is also difficult to make the unstructured ioctl arguments work identically on all systems; for example, consider 64-bit systems with a user-space process running in 32-bit mode. As a result, there is strong pressure to implement miscellaneous control operations by just about any other means. Possible alternatives include embedding commands into the data stream (we will discuss this approach later in this chapter) or using virtual filesystems, either sysfs or driver-specific filesystems. (We will look at sysfs in Chapter 14.) However, the fact remains that ioctl is often the easiest and most straightforward choice for true device operations.

The ioctl driver method has a prototype that differs somewhat from the user-space version:

```
int (*ioctl) (struct inode *inode, struct file *filp,

              unsigned int cmd, unsigned long arg);
```

The inode and filp pointers are the values corresponding to the file descriptor fd passed on by the application and are the same parameters passed to the open method. The cmd argument is passed from the user unchanged, and the optional arg argument is passed in the form of an unsigned long, regardless of whether it was given by the user as an integer or a pointer. If the invoking program doesn't pass a third argument, the arg value received by the driver operation is undefined. Because type checking is disabled on the extra argument, the compiler can't warn you if an invalid argument is passed to ioctl, and any associated bug would be difficult to spot.

As you might imagine, most ioctl implementations consist of a big switch statement that selects the correct behavior according to the cmd argument. Different commands have different numeric values, which are usually given symbolic names to simplify coding. The symbolic name is assigned by a preprocessor definition. Custom drivers usually declare such symbols in their header files; *scull.h* declares them for scull. User programs must, of course, include that header file as well to have access to those symbols.

6.1.1. Choosing the ioctl Commands

Before writing the code for ioctl, you need to choose the numbers that correspond to commands. The first instinct of many programmers is to choose a set of small numbers starting with or 1 and going up from there. There are, however, good reasons for not doing things that way. The ioctl command numbers should be unique across the system in order to prevent errors caused by issuing the right command to the wrong device. Such a mismatch is not unlikely to happen, and a program might find itself trying to change the baud rate of a non-serial-port input stream, such as a FIFO or an audio device. If each ioctl number is unique, the application gets an EINVAL error rather than succeeding in doing something unintended.

To help programmers create unique ioctl command codes, these codes have been split up into several bitfields.

6.2. Blocking I/O

Back in Chapter 3, we looked at how to implement the read and write driver methods. At that point, however, we skipped over one important issue: how does a driver respond if it cannot immediately satisfy the request? A call to read may come when no data is available, but more is expected in the future. Or a process could attempt to write, but your device is not ready to accept the data, because your output buffer is full. The calling process usually does not care about such issues; the programmer simply expects to call read or write and have the call return after the necessary work has been done. So, in such cases, your driver should (by default) *block* the process, putting it to sleep until the request can proceed.

This section shows how to put a process to sleep and wake it up again later on. As usual, however, we have to explain a few concepts first.

6.2.1. Introduction to Sleeping

What does it mean for a process to "sleep"? When a process is put to sleep, it is marked as being in a special state and removed from the scheduler's run queue. Until something comes along to change that state, the process will not be scheduled on any CPU and, therefore, will not run. A sleeping process has been shunted off to the side of the system, waiting for some future event to happen.

Causing a process to sleep is an easy thing for a Linux device driver to do. There are, however, a couple of rules that you must keep in mind to be able to code sleeps in a safe manner.

The first of these rules is: never sleep when you are running in an atomic context. An atomic context is simply a state where multiple steps must be performed without any sort of concurrent access. What that means, with regard to sleeping, is that your driver cannot sleep while holding a spinlock, seqlock, or RCU lock. You also cannot sleep if you have disabled interrupts. It is legal to sleep while holding a semaphore, but you should look very carefully at any code that does so. If code sleeps while holding a semaphore, any other thread waiting for that semaphore also sleeps. So any sleeps that happen while holding semaphores should be short, and you should convince yourself that, by holding the semaphore, you are not blocking the process that will eventually wake you up.

Another thing to remember with sleeping is that, when you wake up, you never know how long your process may have been out of the CPU or what may have changed in the mean time. You also do not usually know if another process may have been sleeping for the same event; that process may wake before you and grab whatever resource you were waiting for. The end result is that you can make no assumptions about the state of the system after you wake up, and you must check to ensure that the condition you were waiting for is, indeed, true.

One other relevant point, of course, is that your process cannot sleep unless it is assured that somebody else, somewhere, will wake it up. The code doing the awakening must also be able to find your process to be able to do its job. Making sure that a wakeup happens is a matter of thinking through your code and knowing, for each sleep, exactly what series of events will bring that sleep to an end. Making it possible for your sleeping process to be found is, instead, accomplished through a data structure called a *wait queue* . A wait queue is just what it sounds like: a list of processes, all waiting for a specific event.

In Linux, a wait queue is managed by means of a "wait queue head," a structure of type wait_queue_head_t, which is defined in *<linux/wait.h>*. A wait queue head can be defined and initialized statically with:

```
DECLARE_WAIT_QUEUE_HEAD(name);
```

or dynamicly as follows:

```
wait_queue_head_t my_queue;

init_waitqueue_head(&my_queue);
```

We will return to the structure of wait queues shortly, but we know enough now to take a first look at sleeping and waking up.

6.2.2. Simple Sleeping

6.3. poll and select

Applications that use nonblocking I/O often use the poll, select, and epoll system calls as well. poll, select, and epoll have essentially the same functionality: each allow a process to determine whether it can read from or write to one or more open files without blocking. These calls can also block a process until any of a given set of file descriptors becomes available for reading or writing. Therefore, they are often used in applications that must use multiple input or output streams without getting stuck on any one of them. The same functionality is offered by multiple functions, because two were implemented in Unix almost at the same time by two different groups: select was introduced in BSD Unix, whereas poll was the System V solution. The epoll call[4] was added in 2.5.45 as a way of making the polling function scale to thousands of file descriptors.

[4] Actually, epoll is a set of three calls that together can be used to achieve the polling functionality. For our purposes, though, we can think of it as a single call.

Support for any of these calls requires support from the device driver. This support (for all three calls) is provided through the driver's poll method. This method has the following prototype:

```
unsigned int (*poll) (struct file *filp, poll_table *wait);
```

The driver method is called whenever the user-space program performs a poll, select, or epoll system call involving a file descriptor associated with the driver. The device method is in charge of these two steps:

1. Call poll_wait on one or more wait queues that could indicate a change in the poll status. If no file descriptors are currently available for I/O, the kernel causes the process to wait on the wait queues for all file descriptors passed to the system call.
2. Return a bit mask describing the operations (if any) that could be immediately performed without blocking.

Both of these operations are usually straightforward and tend to look very similar from one driver to the next. They rely, however, on information that only the driver can provide and, therefore, must be implemented individually by each driver.

The poll_table structure, the second argument to the poll method, is used within the kernel to implement the poll, select, and epoll calls; it is declared in *<linux/poll.h>*, which must be included by the driver source. Driver writers do not need to know anything about its internals and must use it as an opaque object; it is passed to the driver method so that the driver can load it with every wait queue that could wake up the process and change the status of the poll operation. The driver adds a wait queue to the poll_table structure by calling the function poll_wait:

```
void poll_wait (struct file *, wait_queue_head_t *, poll_table *);
```

The second task performed by the poll method is returning the bit mask describing which operations could be completed immediately; this is also straightforward. For example, if the device has data available, a read would complete without sleeping; the poll method should indicate this state of affairs. Several flags (defined via *<linux/poll.h>*) are used to indicate the possible operations:

POLLIN

This bit must be set if the device can be read without blocking.

POLLRDNORM

This bit must be set if "normal" data is available for reading. A readable device returns (POLLIN | POLLRDNORM).

POLLRDBAND

6.4. Asynchronous Notification

Although the combination of blocking and nonblocking operations and the select method are sufficient for querying the device most of the time, some situations aren't efficiently managed by the techniques we've seen so far.

Let's imagine a process that executes a long computational loop at low priority but needs to process incoming data as soon as possible. If this process is responding to new observations available from some sort of data acquisition peripheral, it would like to know immediately when new data is available. This application could be written to call poll regularly to check for data, but, for many situations, there is a better way. By enabling asynchronous notification, this application can receive a signal whenever data becomes available and need not concern itself with polling.

User programs have to execute two steps to enable asynchronous notification from an input file. First, they specify a process as the "owner" of the file. When a process invokes the F_SETOWN command using the fcntl system call, the process ID of the owner process is saved in filp->f_owner for later use. This step is necessary for the kernel to know just whom to notify. In order to actually enable asynchronous notification, the user programs must set the FASYNC flag in the device by means of the F_SETFL fcntl command.

After these two calls have been executed, the input file can request delivery of a SIGIO signal whenever new data arrives. The signal is sent to the process (or process group, if the value is negative) stored in filp->f_owner.

For example, the following lines of code in a user program enable asynchronous notification to the current process for the stdin input file:

```
 signal(SIGIO, &input_handler); /* dummy sample; sigaction(  ) is better */

fcntl(STDIN_FILENO, F_SETOWN, getpid(  ));

oflags = fcntl(STDIN_FILENO, F_GETFL);

fcntl(STDIN_FILENO, F_SETFL, oflags | FASYNC);
```

The program named asynctest in the sources is a simple program that reads stdin as shown. It can be used to test the asynchronous capabilities of scullpipe. The program is similar to cat but doesn't terminate on end-of-file; it responds only to input, not to the absence of input.

Note, however, that not all the devices support asynchronous notification, and you can choose not to offer it. Applications usually assume that the asynchronous capability is available only for sockets and ttys.

There is one remaining problem with input notification. When a process receives a SIGIO, it doesn't know which input file has new input to offer. If more than one file is enabled to asynchronously notify the process of pending input, the application must still resort to poll or select to find out what happened.

6.4.1. The Driver's Point of View

A more relevant topic for us is how the device driver can implement asynchronous signaling. The following list details the sequence of operations from the kernel's point of view:

1. When F_SETOWN is invoked, nothing happens, except that a value is assigned to filp->f_owner.
2. When F_SETFL is executed to turn on FASYNC, the driver's fasync method is called. This method is called whenever the value of FASYNC is changed in filp->f_flags to notify the driver of the change, so it can respond properly. The flag is cleared by default when the file is opened. We'll look at the standard implementation of the driver method later in this section.
3. When data arrives, all the processes registered for asynchronous notification must be sent a SIGIO signal.

6.5. Seeking a Device

One of the last things we need to cover in this chapter is the llseek method, which is useful (for some devices) and easy to implement.

6.5.1. The llseek Implementation

The llseek method implements the lseek and llseek system calls. We have already stated that if the llseek method is missing from the device's operations, the default implementation in the kernel performs seeks by modifying filp->f_pos, the current reading/writing position within the file. Please note that for the lseek system call to work correctly, the read and write methods must cooperate by using and updating the offset item they receive as an argument.

You may need to provide your own llseek method if the seek operation corresponds to a physical operation on the device. A simple example can be seen in the scull driver:

```
 loff_t scull_llseek(struct file *filp, loff_t off, int whence)

{

    struct scull_dev *dev = filp->private_data;

    loff_t newpos;



    switch(whence) {

      case 0: /* SEEK_SET */

        newpos = off;

        break;



      case 1: /* SEEK_CUR */

        newpos = filp->f_pos + off;

        break;



      case 2: /* SEEK_END */

        newpos = dev->size + off;

        break;



      default: /* can't happen */

        return -EINVAL;

    }

    if (newpos < 0) return -EINVAL;

    filp->f_pos = newpos;

    return newpos;

}
```

6.6. Access Control on a Device File

Offering access control is sometimes vital for the reliability of a device node. Not only should unauthorized users not be permitted to use the device (a restriction is enforced by the filesystem permission bits), but sometimes only one authorized user should be allowed to open the device at a time.

The problem is similar to that of using ttys. In that case, the login process changes the ownership of the device node whenever a user logs into the system, in order to prevent other users from interfering with or sniffing the tty data flow. However, it's impractical to use a privileged program to change the ownership of a device every time it is opened just to grant unique access to it.

None of the code shown up to now implements any access control beyond the filesystem permission bits. If the open system call forwards the request to the driver, open succeeds. We now introduce a few techniques for implementing some additional checks.

Every device shown in this section has the same behavior as the bare scull device (that is, it implements a persistent memory area) but differs from scull in access control, which is implemented in the open and release operations.

6.6.1. Single-Open Devices

The brute-force way to provide access control is to permit a device to be opened by only one process at a time (single openness). This technique is best avoided because it inhibits user ingenuity. A user might want to run different processes on the same device, one reading status information while the other is writing data. In some cases, users can get a lot done by running a few simple programs through a shell script, as long as they can access the device concurrently. In other words, implementing a single-open behavior amounts to creating policy, which may get in the way of what your users want to do.

Allowing only a single process to open a device has undesirable properties, but it is also the easiest access control to implement for a device driver, so it's shown here. The source code is extracted from a device called scullsingle.

The scullsingle device maintains an atomic_t variable called scull_s_available; that variable is initialized to a value of one, indicating that the device is indeed available. The open call decrements and tests scull_s_available and refuses access if somebody else already has the device open:

```
 static atomic_t scull_s_available = ATOMIC_INIT(1);



static int scull_s_open(struct inode *inode, struct file *filp)

{

    struct scull_dev *dev = &scull_s_device; /* device information */



    if (! atomic_dec_and_test (&scull_s_available)) {

        atomic_inc(&scull_s_available);

        return -EBUSY; /* already open */

    }



    /* then, everything else is copied from the bare scull device */

    if ( (filp->f_flags & O_ACCMODE) =  = O_WRONLY)

        scull_trim(dev);
```

6.7. Quick Reference

This chapter introduced the following symbols and header files:

#include <linux/ioctl.h>

Declares all the macros used to define ioctl commands. It is currently included by <*linux/fs.h*>.

_IOC_NRBITS

_IOC_TYPEBITS

_IOC_SIZEBITS

_IOC_DIRBITS

The number of bits available for the different bitfields of ioctl commands. There are also four macros that specify the MASKs and four that specify the SHIFTs, but they're mainly for internal use. _IOC_SIZEBITS is an important value to check, because it changes across architectures.

_IOC_NONE

_IOC_READ

_IOC_WRITE

The possible values for the "direction" bitfield. "Read" and "write" are different bits and can be ORed to specify read/write. The values are 0-based.

_IOC(dir,type,nr,size)

_IO(type,nr)

_IOR(type,nr,size)

_IOW(type,nr,size)

_IOWR(type,nr,size)

Macros used to create an ioctl command.

_IOC_DIR(nr)

_IOC_TYPE(nr)

Chapter 7. Time, Delays, and Deferred Work

At this point, we know the basics of how to write a full-featured char module. Real-world drivers, however, need to do more than implement the operations that control a device; they have to deal with issues such as timing, memory management, hardware access, and more. Fortunately, the kernel exports a number of facilities to ease the task of the driver writer. In the next few chapters, we'll describe some of the kernel resources you can use. This chapter leads the way by describing how timing issues are addressed. Dealing with time involves the following tasks, in order of increasing complexity:

- Measuring time lapses and comparing times
- Knowing the current time
- Delaying operation for a specified amount of time
- Scheduling asynchronous functions to happen at a later time

7.1. Measuring Time Lapses

The kernel keeps track of the flow of time by means of timer interrupts. Interrupts are covered in detail in Chapter 10.

Timer interrupts are generated by the system's timing hardware at regular intervals; this interval is programmed at boot time by the kernel according to the value of HZ, which is an architecture-dependent value defined in <*linux/param.h*> or a subplatform file included by it. Default values in the distributed kernel source range from 50 to 1200 ticks per second on real hardware, down to 24 for software simulators. Most platforms run at 100 or 1000 interrupts per second; the popular x86 PC defaults to 1000, although it used to be 100 in previous versions (up to and including 2.4). As a general rule, even if you know the value of HZ, you should never count on that specific value when programming.

It is possible to change the value of HZ for those who want systems with a different clock interrupt frequency. If you change HZ in the header file, you need to recompile the kernel and all modules with the new value. You might want to raise HZ to get a more fine-grained resolution in your asynchronous tasks, if you are willing to pay the overhead of the extra timer interrupts to achieve your goals. Actually, raising HZ to 1000 was pretty common with x86 industrial systems using Version 2.4 or 2.2 of the kernel. With current versions, however, the best approach to the timer interrupt is to keep the default value for HZ, by virtue of our complete trust in the kernel developers, who have certainly chosen the best value. Besides, some internal calculations are currently implemented only for HZ in the range from 12 to 1535 (see <*linux/timex.h*> and RFC-1589).

Every time a timer interrupt occurs, the value of an internal kernel counter is incremented. The counter is initialized to 0 at system boot, so it represents the number of clock ticks since last boot. The counter is a 64-bit variable (even on 32-bit architectures) and is called jiffies_64. However, driver writers normally access the jiffies variable, an unsigned long that is the same as either jiffies_64 or its least significant bits. Using jiffies is usually preferred because it is faster, and accesses to the 64-bit jiffies_64 value are not necessarily atomic on all architectures.

In addition to the low-resolution kernel-managed jiffy mechanism, some CPU platforms feature a high-resolution counter that software can read. Although its actual use varies somewhat across platforms, it's sometimes a very powerful tool.

7.1.1. Using the jiffies Counter

The counter and the utility functions to read it live in <*linux/jiffies.h*>, although you'll usually just include <*linux/sched.h*>, that automatically pulls *jiffies.h* in. Needless to say, both jiffies and jiffies_64 must be considered read-only.

Whenever your code needs to remember the current value of jiffies, it can simply access the unsigned long variable, which is declared as volatile to tell the compiler not to optimize memory reads. You need to read the current counter whenever your code needs to calculate a future time stamp, as shown in the following example:

```
 #include <linux/jiffies.h>

unsigned long j, stamp_1, stamp_half, stamp_n;



j = jiffies;                      /* read the current value */

stamp_1    = j + HZ;              /* 1 second in the future */

stamp_half = j + HZ/2;            /* half a second */

stamp_n    = j + n * HZ / 1000;   /* n milliseconds */
```

This code has no problem with jiffies wrapping around, as long as different values are compared in the right way. Even though on 32-bit platforms the counter wraps around only once every 50 days when HZ is 1000, your code should be prepared to face that event. To compare your cached value (like stamp_1 above) and the current value, you should use one of the following macros:

```
 #include <linux/jiffies.h>
```

7.2. Knowing the Current Time

Kernel code can always retrieve a representation of the current time by looking at the value of jiffies. Usually, the fact that the value represents only the time since the last boot is not relevant to the driver, because its life is limited to the system uptime. As shown, drivers can use the current value of jiffies to calculate time intervals across events (for example, to tell double-clicks from single-clicks in input device drivers or calculate timeouts). In short, looking at jiffies is almost always sufficient when you need to measure time intervals. If you need very precise measurements for short time lapses, processor-specific registers come to the rescue (although they bring in serious portability issues).

It's quite unlikely that a driver will ever need to know the wall-clock time, expressed in months, days, and hours; the information is usually needed only by user programs such as cron and syslogd. Dealing with real-world time is usually best left to user space, where the C library offers better support; besides, such code is often too policy-related to belong in the kernel. There is a kernel function that turns a wall-clock time into a jiffies value, however:

```
#include <linux/time.h>

unsigned long mktime (unsigned int year, unsigned int mon,

                      unsigned int day, unsigned int hour,

                      unsigned int min, unsigned int sec);
```

To repeat: dealing directly with wall-clock time in a driver is often a sign that policy is being implemented and should therefore be questioned.

While you won't have to deal with human-readable representations of the time, sometimes you need to deal with absolute timestamp even in kernel space. To this aim, *<linux/time.h>* exports the do_gettimeofday function. When called, it fills a struct timeval pointer—the same one used in the gettimeofday system call—with the familiar seconds and microseconds values. The prototype for do_gettimeofday is:

```
#include <linux/time.h>

void do_gettimeofday(struct timeval *tv);
```

The source states that do_gettimeofday has "near microsecond resolution," because it asks the timing hardware what fraction of the current jiffy has already elapsed. The precision varies from one architecture to another, however, since it depends on the actual hardware mechanisms in use. For example, some m68knommu processors, Sun3 systems, and other m68k systems cannot offer more than jiffy resolution. Pentium systems, on the other hand, offer very fast and precise subtick measures by reading the timestamp counter described earlier in this chapter.

The current time is also available (though with jiffy granularity) from the xtime variable, a struct timespec value. Direct use of this variable is discouraged because it is difficult to atomically access both the fields. Therefore, the kernel offers the utility function current_kernel_time:

```
#include <linux/time.h>

struct timespec current_kernel_time(void);
```

Code for retrieving the current time in the various ways it is available within the jit ("just in time") module in the source files provided on O'Reilly's FTP site. jit creates a file called */proc/currentime*, which returns the following items in ASCII when read:

- The current jiffies and jiffies_64 values as hex numbers
- The current time as returned by do_gettimeofday
- The timespec returned by current_kernel_time

7.3. Delaying Execution

Device drivers often need to delay the execution of a particular piece of code for a period of time, usually to allow the hardware to accomplish some task. In this section we cover a number of different techniques for achieving delays. The circumstances of each situation determine which technique is best to use; we go over them all, and point out the advantages and disadvantages of each.

One important thing to consider is how the delay you need compares with the clock tick, considering the range of HZ across the various platforms. Delays that are reliably longer than the clock tick, and don't suffer from its coarse granularity, can make use of the system clock. Very short delays typically must be implemented with software loops. In between these two cases lies a gray area. In this chapter, we use the phrase "long" delay to refer to a multiple-jiffy delay, which can be as low as a few milliseconds on some platforms, but is still long as seen by the CPU and the kernel.

The following sections talk about the different delays by taking a somewhat long path from various intuitive but inappropriate solutions to the right solution. We chose this path because it allows a more in-depth discussion of kernel issues related to timing. If you are eager to find the right code, just skim through the section.

7.3.1. Long Delays

Occasionally a driver needs to delay execution for relatively long periods—more than one clock tick. There are a few ways of accomplishing this sort of delay; we start with the simplest technique, then proceed to the more advanced techniques.

7.3.1.1 Busy waiting

If you want to delay execution by a multiple of the clock tick, allowing some slack in the value, the easiest (though not recommended) implementation is a loop that monitors the jiffy counter. The busy-waiting implementation usually looks like the following code, where j1 is the value of jiffies at the expiration of the delay:

```
while (time_before(jiffies, j1))

    cpu_relax(  );
```

The call to cpu_relax invokes an architecture-specific way of saying that you're not doing much with the processor at the moment. On many systems it does nothing at all; on symmetric multithreaded ("hyperthreaded") systems, it may yield the core to the other thread. In any case, this approach should definitely be avoided whenever possible. We show it here because on occasion you might want to run this code to better understand the internals of other code.

So let's look at how this code works. The loop is guaranteed to work because jiffies is declared as volatile by the kernel headers and, therefore, is fetched from memory any time some C code accesses it. Although technically correct (in that it works as designed), this busy loop severely degrades system performance. If you didn't configure your kernel for preemptive operation, the loop completely locks the processor for the duration of the delay; the scheduler never preempts a process that is running in kernel space, and the computer looks completely dead until time j1 is reached. The problem is less serious if you are running a preemptive kernel, because, unless the code is holding a lock, some of the processor's time can be recovered for other uses. Busy waits are still expensive on preemptive systems, however.

Still worse, if interrupts happen to be disabled when you enter the loop, jiffies won't be updated, and the while condition remains true forever. Running a preemptive kernel won't help either, and you'll be forced to hit the big red button.

This implementation of delaying code is available, like the following ones, in the jit module. The */proc/jit** files created by the module delay a whole second each time you read a line of text, and lines are guaranteed to be 20 bytes each. If you want to test the busy-wait code, you can read */proc/jitbusy*, which busy-loops for one second for each line it returns.

Be sure to read, at most, one line (or a few lines) at a time from */proc/jitbusy*. The

7.4. Kernel Timers

Whenever you need to schedule an action to happen later, without blocking the current process until that time arrives, kernel timers are the tool for you. These timers are used to schedule execution of a function at a particular time in the future, based on the clock tick, and can be used for a variety of tasks; for example, polling a device by checking its state at regular intervals when the hardware can't fire interrupts. Other typical uses of kernel timers are turning off the floppy motor or finishing another lengthy shut down operation. In such cases, delaying the return from close would impose an unnecessary (and surprising) cost on the application program. Finally, the kernel itself uses the timers in several situations, including the implementation of schedule_timeout.

A kernel timer is a data structure that instructs the kernel to execute a user-defined function with a user-defined argument at a user-defined time. The implementation resides in <linux/timer.h> and kernel/timer.c and is described in detail in the Section 7.4.2

The functions scheduled to run almost certainly do not run while the process that registered them is executing. They are, instead, run asynchronously. Until now, everything we have done in our sample drivers has run in the context of a process executing system calls. When a timer runs, however, the process that scheduled it could be asleep, executing on a different processor, or quite possibly has exited altogether.

This asynchronous execution resembles what happens when a hardware interrupt happens (which is discussed in detail in Chapter 10). In fact, kernel timers are run as the result of a "software interrupt." When running in this sort of atomic context, your code is subject to a number of constraints. Timer functions must be atomic in all the ways we discussed in Chapter 5, but there are some additional issues brought about by the lack of a process context. We will introduce these constraints now; they will be seen again in several places in later chapters. Repetition is called for because the rules for atomic contexts must be followed assiduously, or the system will find itself in deep trouble.

A number of actions require the context of a process in order to be executed. When you are outside of process context (i.e., in interrupt context), you must observe the following rules:

- No access to user space is allowed. Because there is no process context, there is no path to the user space associated with any particular process.
- The current pointer is not meaningful in atomic mode and cannot be used since the relevant code has no connection with the process that has been interrupted.
- No sleeping or scheduling may be performed. Atomic code may not call schedule or a form of wait_event, nor may it call any other function that could sleep. For example, calling kmalloc(..., GFP_KERNEL) is against the rules. Semaphores also must not be used since they can sleep.

Kernel code can tell if it is running in interrupt context by calling the function in_interrupt( ), which takes no parameters and returns nonzero if the processor is currently running in interrupt context, either hardware interrupt or software interrupt.

A function related to in_interrupt( ) is in_atomic( ). Its return value is nonzero whenever scheduling is not allowed; this includes hardware and software interrupt contexts as well as any time when a spinlock is held. In the latter case, current may be valid, but access to user space is forbidden, since it can cause scheduling to happen. Whenever you are using in_interrupt( ), you should really consider whether in_atomic( ) is what you actually mean. Both functions are declared in <asm/hardirq.h>

One other important feature of kernel timers is that a task can reregister itself to run again at a later time. This is possible because each timer_list structure is unlinked from the list of active timers before being run and can, therefore, be immediately re-linked elsewhere. Although rescheduling the same task over and over might appear to be a pointless operation, it is sometimes useful. For example, it can be used to implement the polling of devices.

It's also worth knowing that in an SMP system, the timer function is executed by the same CPU that registered it

7.5. Tasklets

Another kernel facility related to timing issues is the tasklet mechanism. It is mostly used in interrupt management (we'll see it again in Chapter 10.)

Tasklets resemble kernel timers in some ways. They are always run at interrupt time, they always run on the same CPU that schedules them, and they receive an unsigned long argument. Unlike kernel timers, however, you can't ask to execute the function at a specific time. By scheduling a tasklet, you simply ask for it to be executed at a later time chosen by the kernel. This behavior is especially useful with interrupt handlers, where the hardware interrupt must be managed as quickly as possible, but most of the data management can be safely delayed to a later time. Actually, a tasklet, just like a kernel timer, is executed (in atomic mode) in the context of a "soft interrupt," a kernel mechanism that executes asynchronous tasks with hardware interrupts enabled.

A tasklet exists as a data structure that must be initialized before use. Initialization can be performed by calling a specific function or by declaring the structure using certain macros:

```
#include <linux/interrupt.h>



struct tasklet_struct {

      /* ... */

      void (*func)(unsigned long);

      unsigned long data;

};



void tasklet_init(struct tasklet_struct *t,

      void (*func)(unsigned long), unsigned long data);

DECLARE_TASKLET(name, func, data);

DECLARE_TASKLET_DISABLED(name, func, data);
```

Tasklets offer a number of interesting features:

- A tasklet can be disabled and re-enabled later; it won't be executed until it is enabled as many times as it has been disabled.
- Just like timers, a tasklet can reregister itself.
- A tasklet can be scheduled to execute at normal priority or high priority. The latter group is always executed first.
- Tasklets may be run immediately if the system is not under heavy load but never later than the next timer tick.
- A tasklets can be concurrent with other tasklets but is strictly serialized with respect to itself—the same tasklet never runs simultaneously on more than one processor. Also, as already noted, a tasklet always runs on the same CPU that schedules it.

The jit module includes two files, */proc/jitasklet* and */proc/jitasklethi*, that return the same data as

7.6. Workqueues

Workqueues are, superficially, similar to tasklets; they allow kernel code to request that a function be called at some future time. There are, however, some significant differences between the two, including:

- Tasklets run in software interrupt context with the result that all tasklet code must be atomic. Instead, workqueue functions run in the context of a special kernel process; as a result, they have more flexibility. In particular, workqueue functions can sleep.
- Tasklets always run on the processor from which they were originally submitted. Workqueues work in the same way, by default.
- Kernel code can request that the execution of workqueue functions be delayed for an explicit interval.

The key difference between the two is that tasklets execute quickly, for a short period of time, and in atomic mode, while workqueue functions may have higher latency but need not be atomic. Each mechanism has situations where it is appropriate.

Workqueues have a type of struct workqueue_struct, which is defined in <*linux/workqueue.h*>. A workqueue must be explicitly created before use, using one of the following two functions:

```
struct workqueue_struct *create_workqueue(const char *name);

struct workqueue_struct *create_singlethread_workqueue(const char *name);
```

Each workqueue has one or more dedicated processes ("kernel threads"), which run functions submitted to the queue. If you use create_workqueue, you get a workqueue that has a dedicated thread for each processor on the system. In many cases, all those threads are simply overkill; if a single worker thread will suffice, create the workqueue with create_singlethread_workqueue instead.

To submit a task to a workqueue, you need to fill in a work_struct structure. This can be done at compile time as follows:

```
DECLARE_WORK(name, void (*function)(void *), void *data);
```

Where name is the name of the structure to be declared, function is the function that is to be called from the workqueue, and data is a value to pass to that function. If you need to set up the work_struct structure at runtime, use the following two macros:

```
INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);

PREPARE_WORK(struct work_struct *work, void (*function)(void *), void *data);
```

INIT_WORK does a more thorough job of initializing the structure; you should use it the first time that structure is set up. PREPARE_WORK does almost the same job, but it does not initialize the pointers used to link the work_struct structure into the workqueue. If there is any possibility that the structure may currently be submitted to a workqueue, and you need to change that structure, use PREPARE_WORK rather than INIT_WORK.

There are two functions for submitting work to a workqueue:

```
int queue_work(struct workqueue_struct *queue, struct work_struct *work);

int queue_delayed_work(struct workqueue_struct *queue,

                       struct work_struct *work, unsigned long delay);
```

Either one adds work to the given queue. If queue_delayed_work is used, however, the actual work is not performed until at least delay jiffies have passed. The return value from these functions is 0 if the work was successfully added to the queue; a nonzero result means that this work_struct structure was already waiting in the queue, and was not added a second time.

At some time in the future, the work function will be called with the given data value. The function will be

7.7. Quick Reference

This chapter introduced the following symbols.

7.7.1. Timekeeping

#include <linux/param.h>

HZ

The HZ symbol specifies the number of clock ticks generated per second.

#include <linux/jiffies.h>

volatile unsigned long jiffies

u64 jiffies_64

The jiffies_64 variable is incremented once for each clock tick; thus, it's incremented HZ times per second. Kernel code most often refers to jiffies, which is the same as jiffies_64 on 64-bit platforms and the least significant half of it on 32-bit platforms.

int time_after(unsigned long a, unsigned long b);

int time_before(unsigned long a, unsigned long b);

int time_after_eq(unsigned long a, unsigned long b);

int time_before_eq(unsigned long a, unsigned long b);

These Boolean expressions compare jiffies in a safe way, without problems in case of counter overflow and without the need to access jiffies_64.

u64 get_jiffies_64(void);

Retrieves jiffies_64 without race conditions.

#include <linux/time.h>

unsigned long timespec_to_jiffies(struct timespec *value);

void jiffies_to_timespec(unsigned long jiffies, struct timespec *value);

unsigned long timeval_to_jiffies(struct timeval *value);

Chapter 8. Allocating Memory

Thus far, we have used kmalloc and kfree for the allocation and freeing of memory. The Linux kernel offers a richer set of memory allocation primitives, however. In this chapter, we look at other ways of using memory in device drivers and how to optimize your system's memory resources. We do not get into how the different architectures actually administer memory. Modules are not involved in issues of segmentation, paging, and so on, since the kernel offers a unified memory management interface to the drivers. In addition, we won't describe the internal details of memory management in this chapter, but defer it to Chapter 15.

8.1. The Real Story of kmalloc

The kmalloc allocation engine is a powerful tool and easily learned because of its similarity to malloc. The function is fast (unless it blocks) and doesn't clear the memory it obtains; the allocated region still holds its previous content.[1] The allocated region is also contiguous in physical memory. In the next few sections, we talk in detail about kmalloc, so you can compare it with the memory allocation techniques that we discuss later.

[1] Among other things, this implies that you should explicitly clear any memory that might be exposed to user space or written to a device; otherwise, you risk disclosing information that should be kept private.

8.1.1. The Flags Argument

Remember that the prototype for kmalloc is:

```
#include <linux/slab.h>


void *kmalloc(size_t size, int flags);
```

The first argument to kmalloc is the size of the block to be allocated. The second argument, the allocation flags, is much more interesting, because it controls the behavior of kmalloc in a number of ways.

The most commonly used flag, GFP_KERNEL, means that the allocation (internally performed by calling, eventually, _ _get_free_pages, which is the source of the GFP_ prefix) is performed on behalf of a process running in kernel space. In other words, this means that the calling function is executing a system call on behalf of a process. Using GFP_KERNEL means that kmalloc can put the current process to sleep waiting for a page when called in low-memory situations. A function that allocates memory using GFP_KERNEL must, therefore, be reentrant and cannot be running in atomic context. While the current process sleeps, the kernel takes proper action to locate some free memory, either by flushing buffers to disk or by swapping out memory from a user process.

GFP_KERNEL isn't always the right allocation flag to use; sometimes kmalloc is called from outside a process's context. This type of call can happen, for instance, in interrupt handlers, tasklets, and kernel timers. In this case, the current process should not be put to sleep, and the driver should use a flag of GFP_ATOMIC instead. The kernel normally tries to keep some free pages around in order to fulfill atomic allocation. When GFP_ATOMIC is used, kmalloc can use even the last free page. If that last page does not exist, however, the allocation fails.

Other flags can be used in place of or in addition to GFP_KERNEL and GFP_ATOMIC, although those two cover most of the needs of device drivers. All the flags are defined in *<linux/gfp.h>*, and individual flags are prefixed with a double underscore, such as _ _GFP_DMA. In addition, there are symbols that represent frequently used combinations of flags; these lack the prefix and are sometimes called allocation priorities. The latter include:

GFP_ATOMIC

Used to allocate memory from interrupt handlers and other code outside of a process context. Never sleeps.

GFP_KERNEL

Normal allocation of kernel memory. May sleep.

GFP_USER

Used to allocate memory for user-space pages; it may sleep.

GFP_HIGHUSER

8.2. Lookaside Caches

A device driver often ends up allocating many objects of the same size, over and over. Given that the kernel already maintains a set of memory pools of objects that are all the same size, why not add some special pools for these high-volume objects? In fact, the kernel does implement a facility to create this sort of pool, which is often called a lookaside cache. Device drivers normally do not exhibit the sort of memory behavior that justifies using a lookaside cache, but there can be exceptions; the USB and SCSI drivers in Linux 2.6 use caches.

The cache manager in the Linux kernel is sometimes called the "slab allocator." For that reason, its functions and types are declared in <*linux/slab.h*>. The slab allocator implements caches that have a type of kmem_cache_t; they are created with a call to kmem_cache_create:

```
kmem_cache_t *kmem_cache_create(const char *name, size_t size,
                                size_t offset,
                                unsigned long flags,
                                void (*constructor)(void *, kmem_cache_t *,
                                                    unsigned long flags),
                                void (*destructor)(void *, kmem_cache_t *,
                                                   unsigned long flags));
```

The function creates a new cache object that can host any number of memory areas all of the same size, specified by the size argument. The name argument is associated with this cache and functions as housekeeping information usable in tracking problems; usually, it is set to the name of the type of structure that is cached. The cache keeps a pointer to the name, rather than copying it, so the driver should pass in a pointer to a name in static storage (usually the name is just a literal string). The name cannot contain blanks.

The offset is the offset of the first object in the page; it can be used to ensure a particular alignment for the allocated objects, but you most likely will use 0 to request the default value. flags controls how allocation is done and is a bit mask of the following flags:

SLAB_NO_REAP

Setting this flag protects the cache from being reduced when the system is looking for memory. Setting this flag is normally a bad idea; it is important to avoid restricting the memory allocator's freedom of action unnecessarily.

SLAB_HWCACHE_ALIGN

This flag requires each data object to be aligned to a cache line; actual alignment depends on the cache layout of the host platform. This option can be a good choice if your cache contains items that are frequently accessed on SMP machines. The padding required to achieve cache line alignment can end up wasting significant amounts of memory, however.

SLAB_CACHE_DMA

This flag requires each data object to be allocated in the DMA memory zone.

There is also a set of flags that can be used during the debugging of cache allocations; see *mm/slab.c* for the details. Usually, however, these flags are set globally via a kernel configuration option on systems used for development.

The constructor and destructor arguments to the function are optional functions (but there can be no destructor without a constructor); the former can be used to initialize newly allocated objects, and the latter can be used to

8.3. get_free_page and Friends

If a module needs to allocate big chunks of memory, it is usually better to use a page-oriented technique. Requesting whole pages also has other advantages, which are introduced in Chapter 15.

To allocate pages, the following functions are available:

get_zeroed_page(unsigned int flags);

Returns a pointer to a new page and fills the page with zeros.

_ _get_free_page(unsigned int flags);

Similar to get_zeroed_page, but doesn't clear the page.

_ _get_free_pages(unsigned int flags, unsigned int order);

Allocates and returns a pointer to the first byte of a memory area that is potentially several (physically contiguous) pages long but doesn't zero the area.

The flags argument works in the same way as with kmalloc; usually either GFP_KERNEL or GFP_ATOMIC is used, perhaps with the addition of the _ _GFP_DMA flag (for memory that can be used for ISA direct-memory-access operations) or _ _GFP_HIGHMEM when high memory can be used.[2] order is the base-two logarithm of the number of pages you are requesting or freeing (i.e., log2N). For example, order is 0 if you want one page and 3 if you request eight pages. If order is too big (no contiguous area of that size is available), the page allocation fails. The get_order function, which takes an integer argument, can be used to extract the order from a size (that must be a power of two) for the hosting platform. The maximum allowed value for order is 10 or 11 (corresponding to 1024 or 2048 pages), depending on the architecture. The chances of an order-10 allocation succeeding on anything other than a freshly booted system with a lot of memory are small, however.

[2] Although alloc_pages (described shortly) should really be used for allocating high-memory pages, for reasons we can't really get into until Chapter 15.

If you are curious, */proc/buddyinfo* tells you how many blocks of each order are available for each memory zone on the system.

When a program is done with the pages, it can free them with one of the following functions. The first function is a macro that falls back on the second:

```
void free_page(unsigned long addr);

void free_pages(unsigned long addr, unsigned long order);
```

If you try to free a different number of pages from what you allocated, the memory map becomes corrupted, and the system gets in trouble at a later time.

It's worth stressing that _ _get_free_pages and the other functions can be called at any time, subject to the same rules we saw for kmalloc. The functions can fail to allocate memory in certain circumstances, particularly when GFP_ATOMIC is used. Therefore, the program calling these allocation functions must be prepared to handle an allocation failure.

Although kmalloc(GFP_KERNEL) sometimes fails when there is no available memory, the kernel does its best to fulfill allocation requests. Therefore, it's easy to degrade system responsiveness by allocating too much memory. For example, you can bring the computer down by pushing too much data into a scull device; the system starts crawling while it tries to swap out as much as possible in order to fulfill the kmalloc request. Since every resource is being sucked up by the growing device, the computer is soon rendered unusable; at that point, you can no longer even start a new process to try to deal with the problem. We don't address this issue in scull, since

8.4. vmalloc and Friends

The next memory allocation function that we show you is vmalloc, which allocates a contiguous memory region in the virtual address space. Although the pages are not consecutive in physical memory (each page is retrieved with a separate call to alloc_page), the kernel sees them as a contiguous range of addresses. vmalloc returns 0 (the NULL address) if an error occurs, otherwise, it returns a pointer to a linear memory area of size at least size.

We describe vmalloc here because it is one of the fundamental Linux memory allocation mechanisms. We should note, however, that use of vmalloc is discouraged in most situations. Memory obtained from vmalloc is slightly less efficient to work with, and, on some architectures, the amount of address space set aside for vmalloc is relatively small. Code that uses vmalloc is likely to get a chilly reception if submitted for inclusion in the kernel. If possible, you should work directly with individual pages rather than trying to smooth things over with vmalloc.

That said, let's see how vmalloc works. The prototypes of the function and its relatives (ioremap, which is not strictly an allocation function, is discussed later in this section) are as follows:

```
#include <linux/vmalloc.h>



void *vmalloc(unsigned long size);

void vfree(void * addr);

void *ioremap(unsigned long offset, unsigned long size);

void iounmap(void * addr);
```

It's worth stressing that memory addresses returned by kmalloc and _ get_free_pages are also virtual addresses. Their actual value is still massaged by the MMU (the memory management unit, usually part of the CPU) before it is used to address physical memory.[4] vmalloc is not different in how it uses the hardware, but rather in how the kernel performs the allocation task.

[4] Actually, some architectures define ranges of "virtual" addresses as reserved to address physical memory. When this happens, the Linux kernel takes advantage of the feature, and both the kernel and _ _get_free_pages addresses lie in one of those memory ranges. The difference is transparent to device drivers and other code that is not directly involved with the memory-management kernel subsystem.

The (virtual) address range used by kmalloc and _ _get_free_pages features a one-to-one mapping to physical memory, possibly shifted by a constant PAGE_OFFSET value; the functions don't need to modify the page tables for that address range. The address range used by vmalloc and ioremap, on the other hand, is completely synthetic, and each allocation builds the (virtual) memory area by suitably setting up the page tables.

This difference can be perceived by comparing the pointers returned by the allocation functions. On some platforms (for example, the x86), addresses returned by vmalloc are just beyond the addresses that kmalloc uses. On other platforms (for example, MIPS, IA-64, and x86_64), they belong to a completely different address range. Addresses available for vmalloc are in the range from VMALLOC_START to VMALLOC_END. Both symbols are defined in *<asm/pgtable.h>*.

Addresses allocated by vmalloc can't be used outside of the microprocessor, because they make sense only on top of the processor's MMU. When a driver needs a real physical address (such as a DMA address, used by peripheral hardware to drive the system's bus), you can't easily use vmalloc. The right time to call vmalloc is when you are allocating memory for a large sequential buffer that exists only in software. It's important to note that vmalloc has more overhead than _ _get_free_pages, because it must both retrieve the memory and build the page tables. Therefore, it doesn't make sense to call vmalloc to allocate just one page.

An example of a function in the kernel that uses vmalloc is the create_module system call, which uses vmalloc to get space for the module being created. Code and data of the module are later copied to the allocated space using copy_from_user. In this way, the module appears to be loaded into contiguous memory. You can verify, by looking in */proc/kallsyms*, that kernel symbols exported by modules lie in a different memory range from

8.5. Per-CPU Variables

Per-CPU variables are an interesting 2.6 kernel feature. When you create a per-CPU variable, each processor on the system gets its own copy of that variable. This may seem like a strange thing to want to do, but it has its advantages. Access to per-CPU variables requires (almost) no locking, because each processor works with its own copy. Per-CPU variables can also remain in their respective processors' caches, which leads to significantly better performance for frequently updated quantities.

A good example of per-CPU variable use can be found in the networking subsystem. The kernel maintains no end of counters tracking how many of each type of packet was received; these counters can be u pdated thousands of times per second. Rather than deal with the caching and locking issues, the networking developers put the statistics counters into per-CPU variables. Updates are now lockless and fast. On the rare occasion that user space requests to see the values of the counters, it is a simple matter to add up each processor's version and return the total.

The declarations for per-CPU variables can be found in *<linux/percpu.h>*. To create a per-CPU variable at compile time, use this macro:

```
DEFINE_PER_CPU(type, name);
```

If the variable (to be called name) is an array, include the dimension information with the type. Thus, a per-CPU array of three integers would be created with:

```
DEFINE_PER_CPU(int[3], my_percpu_array);
```

Per-CPU variables can be manipulated without explicit locking—almost. Remember that the 2.6 kernel is preemptible; it would not do for a processor to be preempted in the middle of a critical section that modifies a per-CPU variable. It also would not be good if your process were to be moved to another processor in the middle of a per-CPU variable access. For this reason, you must explicitly use the get_cpu_var macro to access the current processor's copy of a given variable, and call put_cpu_var when you are done. The call to get_cpu_var returns an lvalue for the current processor's version of the variable and disables preemption. Since an lvalue is returned, it can be assigned to or operated on directly. For example, one counter in the networking code is incremented with these two statements:

```
get_cpu_var(sockets_in_use)++;

put_cpu_var(sockets_in_use);
```

You can access another processor's copy of the variable with:

```
per_cpu(variable, int cpu_id);
```

If you write code that involves processors reaching into each other's per-CPU variables, you, of course, have to implement a locking scheme that makes that access safe.

Dynamically allocated per-CPU variables are also possible. These variables can be allocated with:

```
void *alloc_percpu(type);

void *_ _alloc_percpu(size_t size, size_t align);
```

In most cases, alloc_percpu does the job; you can call _ _alloc_percpu in cases where a particular alignment is required. In either case, a per-CPU variable can be returned to the system with free_percpu. Access to a dynamically allocated per-CPU variable is done via per_cpu_ptr:

```
per_cpu_ptr(void *per_cpu_var, int cpu_id);
```

This macro returns a pointer to the version of per_cpu_var corresponding to the given cpu_id. If you are simply reading another CPU's version of the variable, you can dereference that pointer and be done with it. If, however, you are manipulating the current processor's version, you probably need to ensure that you cannot be moved out of that processor first. If the entirety of your access to the per-CPU variable happens with a spinlock held, all is well. Usually, however, you need to use get_cpu to block preemption while working with the variable. Thus, code using dynamic per-CPU variables tends to look like this:

```
int cpu;



cpu = get_cpu( );
```

8.6. Obtaining Large Buffers

As we have noted in previous sections, allocations of large, contiguous memory buffers are prone to failure. System memory fragments over time, and chances are that a truly large region of memory will simply not be available. Since there are usually ways of getting the job done without huge buffers, the kernel developers have not put a high priority on making large allocations work. Before you try to obtain a large memory area, you should really consider the alternatives. By far the best way of performing large I/O operations is through scatter/gather operations, which we discuss in Chapter 15.

8.6.1. Acquiring a Dedicated Buffer at Boot Time

If you really need a huge buffer of physically contiguous memory, the best approach is often to allocate it by requesting memory at boot time. Allocation at boot time is the only way to retrieve consecutive memory pages while bypassing the limits imposed by _ _get_free_pages on the buffer size, both in terms of maximum allowed size and limited choice of sizes. Allocating memory at boot time is a "dirty" technique, because it bypasses all memory management policies by reserving a private memory pool. This technique is inelegant and inflexible, but it is also the least prone to failure. Needless to say, a module can't allocate memory at boot time; only drivers directly linked to the kernel can do that.

One noticeable problem with boot-time allocation is that it is not a feasible option for the average user, since this mechanism is available only for code linked in the kernel image. A device driver using this kind of allocation can be installed or replaced only by rebuilding the kernel and rebooting the computer.

When the kernel is booted, it gains access to all the physical memory available in the system. It then initializes each of its subsystems by calling that subsystem's initialization function, allowing initialization code to allocate a memory buffer for private use by reducing the amount of RAM left for normal system operation.

Boot-time memory allocation is performed by calling one of these functions:

```
#include <linux/bootmem.h>

void *alloc_bootmem(unsigned long size);

void *alloc_bootmem_low(unsigned long size);

void *alloc_bootmem_pages(unsigned long size);

void *alloc_bootmem_low_pages(unsigned long size);
```

The functions allocate either whole pages (if they end with _pages) or non-page-aligned memory areas. The allocated memory may be high memory unless one of the _low versions is used. If you are allocating this buffer for a device driver, you probably want to use it for DMA operations, and that is not always possible with high memory; thus, you probably want to use one of the _low variants.

It is rare to free memory allocated at boot time; you will almost certainly be unable to get it back later if you want it. There is an interface to free this memory, however:

```
void free_bootmem(unsigned long addr, unsigned long size);
```

Note that partial pages freed in this manner are not returned to the system—but, if you are using this technique, you have probably allocated a fair number of whole pages to begin with.

If you must use boot-time allocation, you need to link your driver directly into the kernel. See the files in the kernel source under *Documentation/kbuild* for more information on how this should be done.

8.7. Quick Reference

The functions and symbols related to memory allocation are:

#include <linux/slab.h>

void *kmalloc(size_t size, int flags);

void kfree(void *obj);

The most frequently used interface to memory allocation.

#include <linux/mm.h>

GFP_USER

GFP_KERNEL

GFP_NOFS

GFP_NOIO

GFP_ATOMIC

Flags that control how memory allocations are performed, from the least restrictive to the most. The GFP_USER and GFP_KERNEL priorities allow the current process to be put to sleep to satisfy the request. GFP_NOFS and GFP_NOIO disable filesystem operations and all I/O operations, respectively, while GFP_ATOMIC allocations cannot sleep at all.

__GFP_DMA

__GFP_HIGHMEM

__GFP_COLD

__GFP_NOWARN

__GFP_HIGH

__GFP_REPEAT

__GFP_NOFAIL

Chapter 9. Communicating with Hardware

Although playing with scull and similar toys is a good introduction to the software interface of a Linux device driver, implementing a real device requires hardware. The driver is the abstraction layer between software concepts and hardware circuitry; as such, it needs to talk with both of them. Up until now, we have examined the internals of software concepts; this chapter completes the picture by showing you how a driver can access I/O ports and I/O memory while being portable across Linux platforms.

This chapter continues in the tradition of staying as independent of specific hardware as possible. However, where specific examples are needed, we use simple digital I/O ports (such as the standard PC parallel port) to show how the I/O instructions work and normal frame-buffer video memory to show memory-mapped I/O.

We chose simple digital I/O because it is the easiest form of an input/output port. Also, the parallel port implements raw I/O and is available in most computers: data bits written to the device appear on the output pins, and voltage levels on the input pins are directly accessible by the processor. In practice, you have to connect LEDs or a printer to the port to actually see the results of a digital I/O operation, but the underlying hardware is extremely easy to use.

9.1. I/O Ports and I/O Memory

Every peripheral device is controlled by writing and reading its registers. Most of the time a device has several registers, and they are accessed at consecutive addresses, either in the memory address space or in the I/O address space.

At the hardware level, there is no conceptual difference between memory regions and I/O regions: both of them are accessed by asserting electrical signals on the address bus and control bus (i.e., the read and write signals) [1] and by reading from or writing to the data bus.

[1] Not all computer platforms use a read and a write signal; some have different means to address external circuits. The difference is irrelevant at software level, however, and we'll assume all have read and write to simplify the discussion.

While some CPU manufacturers implement a single address space in their chips, others decided that peripheral devices are different from memory and, therefore, deserve a separate address space. Some processors (most notably the x86 family) have separate read and write electrical lines for I/O ports and special CPU instructions to access ports.

Because peripheral devices are built to fit a peripheral bus, and the most popular I/O buses are modeled on the personal computer, even processors that do not have a separate address space for I/O ports must fake reading and writing I/O ports when accessing some peripheral devices, usually by means of external chipsets or extra circuitry in the CPU core. The latter solution is common within tiny processors meant for embedded use.

For the same reason, Linux implements the concept of I/O ports on all computer platforms it runs on, even on platforms where the CPU implements a single address space. The implementation of port access sometimes depends on the specific make and model of the host computer (because different models use different chipsets to map bus transactions into memory address space).

Even if the peripheral bus has a separate address space for I/O ports, not all devices map their registers to I/O ports. While use of I/O ports is common for ISA peripheral boards, most PCI devices map registers into a memory address region. This I/O memory approach is generally preferred, because it doesn't require the use of special-purpose processor instructions; CPU cores access memory much more efficiently, and the compiler has much more freedom in register allocation and addressing-mode selection when accessing memory.

9.1.1. I/O Registers and Conventional Memory

Despite the strong similarity between hardware registers and memory, a programmer accessing I/O registers must be careful to avoid being tricked by CPU (or compiler) optimizations that can modify the expected I/O behavior.

The main difference between I/O registers and RAM is that I/O operations have side effects, while memory operations have none: the only effect of a memory write is storing a value to a location, and a memory read returns the last value written there. Because memory access speed is so critical to CPU performance, the no-side-effects case has been optimized in several ways: values are cached and read/write instructions are reordered.

The compiler can cache data values into CPU registers without writing them to memory, and even if it stores them, both write and read operations can operate on cache memory without ever reaching physical RAM. Reordering can also happen both at the compiler level and at the hardware level: often a sequence of instructions can be executed more quickly if it is run in an order different from that which appears in the program text, for example, to prevent interlocks in the RISC pipeline. On CISC processors, operations that take a significant amount of time can be executed concurrently with other, quicker ones.

These optimizations are transparent and benign when applied to conventional memory (at least on uniprocessor systems), but they can be fatal to correct I/O operations, because they interfere with those "side effects" that are the main reason why a driver accesses I/O registers. The processor cannot anticipate a situation in which some other process (running on a separate processor, or something happening inside an I/O controller) depends on the order of memory access. The compiler or the CPU may just try to outsmart you and reorder the operations you request; the result can be strange errors that are very difficult to debug. Therefore, a driver must ensure that no

9.2. Using I/O Ports

I/O ports are the means by which drivers communicate with many devices, at least part of the time. This section covers the various functions available for making use of I/O ports; we also touch on some portability issues.

9.2.1. I/O Port Allocation

As you might expect, you should not go off and start pounding on I/O ports without first ensuring that you have exclusive access to those ports. The kernel provides a registration interface that allows your driver to claim the ports it needs. The core function in that interface is request_region:

```
#include <linux/ioport.h>

struct resource *request_region(unsigned long first, unsigned long n,
                                const char *name);
```

This function tells the kernel that you would like to make use of n ports, starting with first. The name parameter should be the name of your device. The return value is non-NULL if the allocation succeeds. If you get NULL back from request_region, you will not be able to use the desired ports.

All port allocations show up in */proc/ioports*. If you are unable to allocate a needed set of ports, that is the place to look to see who got there first.

When you are done with a set of I/O ports (at module unload time, perhaps), they should be returned to the system with:

```
void release_region(unsigned long start, unsigned long n);
```

There is also a function that allows your driver to check to see whether a given set of I/O ports is available:

```
int check_region(unsigned long first, unsigned long n);
```

Here, the return value is a negative error code if the given ports are not available. This function is deprecated because its return value provides no guarantee of whether an allocation would succeed; checking and later allocating are not an atomic operation. We list it here because several drivers are still using it, but you should always use request_region, which performs the required locking to ensure that the allocation is done in a safe, atomic manner.

9.2.2. Manipulating I/O ports

After a driver has requested the range of I/O ports it needs to use in its activities, it must read and/or write to those ports. To this end, most hardware differentiates between 8-bit, 16-bit, and 32-bit ports. Usually you can't mix them like you normally do with system memory access.[2]

[2] Sometimes I/O ports are arranged like memory, and you can (for example) bind two 8-bit writes into a single 16-bit operation. This applies, for instance, to PC video boards. But generally, you can't count on this feature.

A C program, therefore, must call different functions to access different size ports. As suggested in the previous section, computer architectures that support only memory-mapped I/O registers fake port I/O by remapping port addresses to memory addresses, and the kernel hides the details from the driver in order to ease portability. The Linux kernel headers (specifically, the architecture-dependent header *<asm/io.h>*) define the following inline functions to access I/O ports:

unsigned inb(unsigned port);

void outb(unsigned char byte, unsigned port);

Read or write byte ports (eight bits wide). The port argument is defined as unsigned long for some platforms and unsigned short for others. The return type of inb is also different across architectures.

9.3. An I/O Port Example

The sample code we use to show port I/O from within a device driver acts on general-purpose digital I/O ports; such ports are found in most computer systems.

A digital I/O port, in its most common incarnation, is a byte-wide I/O location, either memory-mapped or port-mapped. When you write a value to an output location, the electrical signal seen on output pins is changed according to the individual bits being written. When you read a value from the input location, the current logic level seen on input pins is returned as individual bit values.

The actual implementation and software interface of such I/O ports varies from system to system. Most of the time, I/O pins are controlled by two I/O locations: one that allows selecting what pins are used as input and what pins are used as output and one in which you can actually read or write logic levels. Sometimes, however, things are even simpler, and the bits are hardwired as either input or output (but, in this case, they're no longer called "general-purpose I/O"); the parallel port found on all personal computers is one such not-so-general-purpose I/O port. Either way, the I/O pins are usable by the sample code we introduce shortly.

9.3.1. An Overview of the Parallel Port

Because we expect most readers to be using an x86 platform in the form called "personal computer," we feel it is worth explaining how the PC parallel port is designed. The parallel port is the peripheral interface of choice for running digital I/O sample code on a personal computer. Although most readers probably have parallel port specifications available, we summarize them here for your convenience.

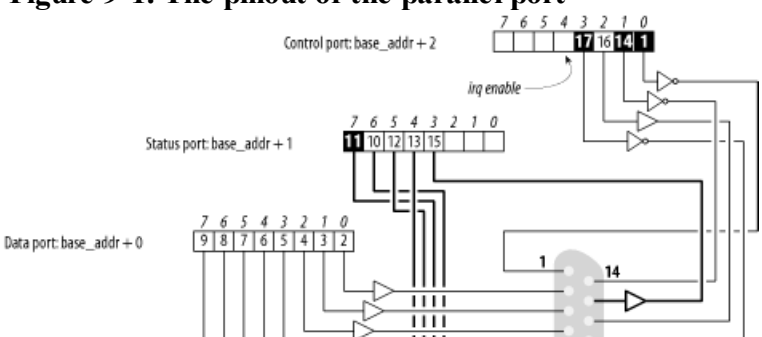
The parallel interface, in its minimal configuration (we overlook the ECP and EPP modes) is made up of three 8-bit ports. The PC standard starts the I/O ports for the first parallel interface at 0x378 and for the second at 0x278. The first port is a bidirectional data register; it connects directly to pins 2-9 on the physical connector. The second port is a read-only status register; when the parallel port is being used for a printer, this register reports several aspects of printer status, such as being online, out of paper, or busy. The third port is an output-only control register, which, among other things, controls whether interrupts are enabled.

The signal levels used in parallel communications are standard transistor-transistor logic (TTL) levels: 0 and 5 volts, with the logic threshold at about 1.2 volts. You can count on the ports at least meeting the standard TTL LS current ratings, although most modern parallel ports do better in both current and voltage ratings.

The parallel connector is not isolated from the computer's internal circuitry, which is useful if you want to connect logic gates directly to the port. But you have to be careful to do the wiring correctly; the parallel port circuitry is easily damaged when you play with your own custom circuitry, unless you add optoisolators to your circuit. You can choose to use plug-in parallel ports if you fear you'll damage your motherboard.

The bit specifications are outlined in Figure 9-1. You can access 12 output bits and 5 input bits, some of which are logically inverted over the course of their signal path. The only bit with no associated signal pin is bit 4 (0x10) of port 2, which enables interrupts from the parallel port. We use this bit as part of our implementation of an interrupt handler in Chapter 10.

Figure 9-1. The pinout of the parallel port



9.4. Using I/O Memory

Despite the popularity of I/O ports in the x86 world, the main mechanism used to communicate with devices is through memory-mapped registers and device memory. Both are called I/O memory because the difference between registers and memory is transparent to software.

I/O memory is simply a region of RAM-like locations that the device makes available to the processor over the bus. This memory can be used for a number of purposes, such as holding video data or Ethernet packets, as well as implementing device registers that behave just like I/O ports (i.e., they have side effects associated with reading and writing them).

The way to access I/O memory depends on the computer architecture, bus, and device being used, although the principles are the same everywhere. The discussion in this chapter touches mainly on ISA and PCI memory, while trying to convey general information as well. Although access to PCI memory is introduced here, a thorough discussion of PCI is deferred to Chapter 12.

Depending on the computer platform and bus being used, I/O memory may or may not be accessed through page tables. When access passes though page tables, the kernel must first arrange for the physical address to be visible from your driver, and this usually means that you must call ioremap before doing any I/O. If no page tables are needed, I/O memory locations look pretty much like I/O ports, and you can just read and write to them using proper wrapper functions.

Whether or not ioremap is required to access I/O memory, direct use of pointers to I/O memory is discouraged. Even though (as introduced in Section 9.1) I/O memory is addressed like normal RAM at hardware level, the extra care outlined in the Section 9.1.1 suggests avoiding normal pointers. The wrapper functions used to access I/O memory are safe on all platforms and are optimized away whenever straight pointer dereferencing can perform the operation.

Therefore, even though dereferencing a pointer works (for now) on the x86, failure to use the proper macros hinders the portability and readability of the driver.

9.4.1. I/O Memory Allocation and Mapping

I/O memory regions must be allocated prior to use. The interface for allocation of memory regions (defined in *<linux/ioport.h>*) is:

```
struct resource *request_mem_region(unsigned long start, unsigned long len,
                                    char *name);
```

This function allocates a memory region of len bytes, starting at start. If all goes well, a non-NULL pointer is returned; otherwise the return value is NULL. All I/O memory allocations are listed in */proc/iomem*.

Memory regions should be freed when no longer needed:

```
void release_mem_region(unsigned long start, unsigned long len);
```

There is also an old function for checking I/O memory region availability:

```
int check_mem_region(unsigned long start, unsigned long len);
```

But, as with check_region, this function is unsafe and should be avoided.

Allocation of I/O memory is not the only required step before that memory may be accessed. You must also ensure that this I/O memory has been made accessible to the kernel. Getting at I/O memory is not just a matter of dereferencing a pointer; on many systems, I/O memory is not directly accessible in this way at all. So a mapping must be set up first. This is the role of the ioremap function, introduced in Section 8.4 in Chapter 8. The function is designed specifically to assign virtual addresses to I/O memory regions.

Once equipped with ioremap (and iounmap), a device driver can access any I/O memory address, whether or not it is directly mapped to virtual address space. Remember, though, that the addresses returned from ioremap should not be dereferenced directly; instead, accessor functions provided by the kernel should be used. Before we get into those functions, we'd better review the ioremap prototypes and introduce a few details that we

9.5. Quick Reference

This chapter introduced the following symbols related to hardware management:

```
#include <linux/kernel.h>

void barrier(void)
```

This "software" memory barrier requests the compiler to consider all memory volatile across this instruction.

```
#include <asm/system.h>

void rmb(void);

void read_barrier_depends(void);

void wmb(void);

void mb(void);
```

Hardware memory barriers. They request the CPU (and the compiler) to checkpoint all memory reads, writes, or both across this instruction.

```
#include <asm/io.h>

unsigned inb(unsigned port);

void outb(unsigned char byte, unsigned port);

unsigned inw(unsigned port);

void outw(unsigned short word, unsigned port);

unsigned inl(unsigned port);

void outl(unsigned doubleword, unsigned port);
```

Functions that are used to read and write I/O ports. They can also be called by user-space programs, provided they have the right privileges to access ports.

```
unsigned inb_p(unsigned port);

...
```

Chapter 10. Interrupt Handling

Although some devices can be controlled using nothing but their I/O regions, most real devices are a bit more complicated than that. Devices have to deal with the external world, which often includes things such as spinning disks, moving tape, wires to distant places, and so on. Much has to be done in a time frame that is different from, and far slower than, that of the processor. Since it is almost always undesirable to have the processor wait on external events, there must be a way for a device to let the processor know when something has happened.

That way, of course, is interrupts. An interrupt is simply a signal that the hardware can send when it wants the processor's attention. Linux handles interrupts in much the same way that it handles signals in user space. For the most part, a driver need only register a handler for its device's interrupts, and handle them properly when they arrive. Of course, underneath that simple picture there is some complexity; in particular, interrupt handlers are somewhat limited in the actions they can perform as a result of how they are run.

It is difficult to demonstrate the use of interrupts without a real hardware device to generate them. Thus, the sample code used in this chapter works with the parallel port. Such ports are starting to become scarce on modern hardware, but, with luck, most people are still able to get their hands on a system with an available port. We'll be working with the short module from the previous chapter; with some small additions it can generate and handle interrupts from the parallel port. The module's name, short, actually means short int (it is C, isn't it?), to remind us that it handles interrupts.

Before we get into the topic, however, it is time for one cautionary note. Interrupt handlers, by their nature, run concurrently with other code. Thus, they inevitably raise issues of concurrency and contention for data structures and hardware. If you succumbed to the temptation to pass over the discussion in Chapter 5, we understand. But we also recommend that you turn back and have another look now. A solid understanding of concurrency control techniques is vital when working with interrupts.

10.1. Preparing the Parallel Port

Although the parallel interface is simple, it can trigger interrupts. This capability is used by the printer to notify the lp driver that it is ready to accept the next character in the buffer.

Like most devices, the parallel port doesn't actually generate interrupts before it's instructed to do so; the parallel standard states that setting bit 4 of port 2 (0x37a, 0x27a, or whatever) enables interrupt reporting. A simple outb call to set the bit is performed by short at module initialization.

Once interrupts are enabled, the parallel interface generates an interrupt whenever the electrical signal at pin 10 (the so-called ACK bit) changes from low to high. The simplest way to force the interface to generate interrupts (short of hooking up a printer to the port) is to connect pins 9 and 10 of the parallel connector. A short length of wire inserted into the appropriate holes in the parallel port connector on the back of your system creates this connection. The pinout of the parallel port is shown in Figure 9-1.

Pin 9 is the most significant bit of the parallel data byte. If you write binary data to */dev/short0*, you generate several interrupts. Writing ASCII text to the port won't generate any interrupts, though, because the ASCII character set has no entries with the top bit set.

If you'd rather avoid wiring pins together, but you do have a printer at hand, you can run the sample interrupt handler using a real printer, as shown later. However, note that the probing functions we introduce depend on the jumper between pin 9 and 10 being in place, and you need it to experiment with probing using our code.

10.2. Installing an Interrupt Handler

If you want to actually "see" interrupts being generated, writing to the hardware device isn't enough; a software handler must be configured in the system. If the Linux kernel hasn't been told to expect your interrupt, it simply acknowledges and ignores it.

Interrupt lines are a precious and often limited resource, particularly when there are only 15 or 16 of them. The kernel keeps a registry of interrupt lines, similar to the registry of I/O ports. A module is expected to request an interrupt channel (or IRQ, for interrupt request) before using it and to release it when finished. In many situations, modules are also expected to be able to share interrupt lines with other drivers, as we will see. The following functions, declared in <*linux/interrupt.h*>, implement the interrupt registration interface:

```
int request_irq(unsigned int irq,

                irqreturn_t (*handler)(int, void *, struct pt_regs *),

                unsigned long flags,

                const char *dev_name,

                void *dev_id);



void free_irq(unsigned int irq, void *dev_id);
```

The value returned from request_irq to the requesting function is either 0 to indicate success or a negative error code, as usual. It's not uncommon for the function to return -EBUSY to signal that another driver is already using the requested interrupt line. The arguments to the functions are as follows:

unsigned int irq

The interrupt number being requested.

irqreturn_t (*handler)(int, void *, struct pt_regs *)

The pointer to the handling function being installed. We discuss the arguments to this function and its return value later in this chapter.

unsigned long flags

As you might expect, a bit mask of options (described later) related to interrupt management.

const char *dev_name

The string passed to request_irq is used in */proc/interrupts* to show the owner of the interrupt (see the next section).

void *dev_id

Pointer used for shared interrupt lines. It is a unique identifier that is used when the interrupt line is freed and that may also be used by the driver to point to its own private data area (to identify which device is interrupting). If the interrupt is not shared, dev_id can be set to NULL, but it a good idea anyway to use this item to point to the device structure. We'll see a practical use for dev_id in Section 10.3.

The bits that can be set in flags are as follows:

10.3. Implementing a Handler

So far, we've learned to register an interrupt handler but not to write one. Actually, there's nothing unusual about a handler—it's ordinary C code.

The only peculiarity is that a handler runs at interrupt time and, therefore, suffers some restrictions on what it can do. These restrictions are the same as those we saw with kernel timers. A handler can't transfer data to or from user space, because it doesn't execute in the context of a process. Handlers also cannot do anything that would sleep, such as calling wait_event, allocating memory with anything other than GFP_ATOMIC, or locking a semaphore. Finally, handlers cannot call schedule.

The role of an interrupt handler is to give feedback to its device about interrupt reception and to read or write data according to the meaning of the interrupt being serviced. The first step usually consists of clearing a bit on the interface board; most hardware devices won't generate other interrupts until their "interrupt-pending" bit has been cleared. Depending on how your hardware works, this step may need to be performed last instead of first; there is no catch-all rule here. Some devices don't require this step, because they don't have an "interrupt-pending" bit; such devices are a minority, although the parallel port is one of them. For that reason, short does not have to clear such a bit.

A typical task for an interrupt handler is awakening processes sleeping on the device if the interrupt signals the event they're waiting for, such as the arrival of new data.

To stick with the frame grabber example, a process could acquire a sequence of images by continuously reading the device; the read call blocks before reading each frame, while the interrupt handler awakens the process as soon as each new frame arrives. This assumes that the grabber interrupts the processor to signal successful arrival of each new frame.

The programmer should be careful to write a routine that executes in a minimum amount of time, independent of its being a fast or slow handler. If a long computation needs to be performed, the best approach is to use a tasklet or workqueue to schedule computation at a safer time (we'll look at how work can be deferred in this manner in Section 10.4.)

Our sample code in short responds to the interrupt by calling do_gettimeofday and printing the current time into a page-sized circular buffer. It then awakens any reading process, because there is now data available to be read.

```
irqreturn_t short_interrupt(int irq, void *dev_id, struct pt_regs *regs)

{

    struct timeval tv;

    int written;



    do_gettimeofday(&tv);



        /* Write a 16 byte record. Assume PAGE_SIZE is a multiple of 16 */

    written = sprintf((char *)short_head,"%08u.%06u\n",

            (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));

    BUG_ON(written != 16);

    short_incr_bp(&short_head, written);

    wake_up_interruptible(&short_queue); /* awake any reading process */

    return IRQ_HANDLED;
```

10.4. Top and Bottom Halves

One of the main problems with interrupt handling is how to perform lengthy tasks within a handler. Often a substantial amount of work must be done in response to a device interrupt, but interrupt handlers need to finish up quickly and not keep interrupts blocked for long. These two needs (work and speed) conflict with each other, leaving the driver writer in a bit of a bind.

Linux (along with many other systems) resolves this problem by splitting the interrupt handler into two halves. The so-called *top half* is the routine that actually responds to the interrupt—the one you register with request_irq. The *bottom half* is a routine that is scheduled by the top half to be executed later, at a safer time. The big difference between the top-half handler and the bottom half is that all interrupts are enabled during execution of the bottom half—that's why it runs at a safer time. In the typical scenario, the top half saves device data to a device-specific buffer, schedules its bottom half, and exits: this operation is very fast. The bottom half then performs whatever other work is required, such as awakening processes, starting up another I/O operation, and so on. This setup permits the top half to service a new interrupt while the bottom half is still working.

Almost every serious interrupt handler is split this way. For instance, when a network interface reports the arrival of a new packet, the handler just retrieves the data and pushes it up to the protocol layer; actual processing of the packet is performed in a bottom half.

The Linux kernel has two different mechanisms that may be used to implement bottom-half processing, both of which were introduced in Chapter 7. Tasklets are often the preferred mechanism for bottom-half processing; they are very fast, but all tasklet code must be atomic. The alternative to tasklets is workqueues, which may have a higher latency but that are allowed to sleep.

The following discussion works, once again, with the short driver. When loaded with a module option, short can be told to do interrupt processing in a top/bottom-half mode with either a tasklet or workqueue handler. In this case, the top half executes quickly; it simply remembers the current time and schedules the bottom half processing. The bottom half is then charged with encoding this time and awakening any user processes that may be waiting for data.

10.4.1. Tasklets

Remember that tasklets are a special function that may be scheduled to run, in software interrupt context, at a system-determined safe time. They may be scheduled to run multiple times, but tasklet scheduling is not cumulative; the tasklet runs only once, even if it is requested repeatedly before it is launched. No tasklet ever runs in parallel with itself, since they run only once, but tasklets can run in parallel with other tasklets on SMP systems. Thus, if your driver has multiple tasklets, they must employ some sort of locking to avoid conflicting with each other.

Tasklets are also guaranteed to run on the same CPU as the function that first schedules them. Therefore, an interrupt handler can be secure that a tasklet does not begin executing before the handler has completed. However, another interrupt can certainly be delivered while the tasklet is running, so locking between the tasklet and the interrupt handler may still be required.

Tasklets must be declared with the DECLARE_TASKLET macro:

```
DECLARE_TASKLET(name, function, data);
```

name is the name to be given to the tasklet, function is the function that is called to execute the tasklet (it takes one unsigned long argument and returns void), and data is an unsigned long value to be passed to the tasklet function.

The short driver declares its tasklet as follows:

```
void short_do_tasklet(unsigned long);

DECLARE_TASKLET(short_tasklet, short_do_tasklet, 0);
```

The function tasklet_schedule is used to schedule a tasklet for running. If short is loaded with tasklet=1, it installs a different interrupt handler that saves data and schedules the tasklet as follows:

```
irqreturn_t short_tl_interrupt(int irq, void *dev_id, struct pt_regs *regs)
```

10.5. Interrupt Sharing

The notion of an IRQ conflict is almost synonymous with the PC architecture. In the past, IRQ lines on the PC have not been able to serve more than one device, and there have never been enough of them. As a result, frustrated users have often spent much time with their computer case open, trying to find a way to make all of their peripherals play well together.

Modern hardware, of course, has been designed to allow the sharing of interrupts; the PCI bus requires it. Therefore, the Linux kernel supports interrupt sharing on all buses, even those (such as the ISA bus) where sharing has traditionally not been supported. Device drivers for the 2.6 kernel should be written to work with shared interrupts if the target hardware can support that mode of operation. Fortunately, working with shared interrupts is easy, most of the time.

10.5.1. Installing a Shared Handler

Shared interrupts are installed through request_irq just like nonshared ones, but there are two differences:

- The SA_SHIRQ bit must be specified in the flags argument when requesting the interrupt.
- The dev_id argument must be unique. Any pointer into the module's address space will do, but dev_id definitely cannot be set to NULL.

The kernel keeps a list of shared handlers associated with the interrupt, and dev_id can be thought of as the signature that differentiates between them. If two drivers were to register NULL as their signature on the same interrupt, things might get mixed up at unload time, causing the kernel to oops when an interrupt arrived. For this reason, modern kernels complain loudly if passed a NULL dev_id when registering shared interrupts. When a shared interrupt is requested, request_irq succeeds if one of the following is true:

- The interrupt line is free.
- All handlers already registered for that line have also specified that the IRQ is to be shared.

Whenever two or more drivers are sharing an interrupt line and the hardware interrupts the processor on that line, the kernel invokes every handler registered for that interrupt, passing each its own dev_id. Therefore, a shared handler must be able to recognize its own interrupts and should quickly exit when its own device has not interrupted. Be sure to return IRQ_NONE whenever your handler is called and finds that the device is not interrupting.

If you need to probe for your device before requesting the IRQ line, the kernel can't help you. No probing function is available for shared handlers. The standard probing mechanism works if the line being used is free, but if the line is already held by another driver with sharing capabilities, the probe fails, even if your driver would have worked perfectly. Fortunately, most hardware designed for interrupt sharing is also able to tell the processor which interrupt it is using, thus eliminating the need for explicit probing.

Releasing the handler is performed in the normal way, using free_irq. Here the dev_id argument is used to select the correct handler to release from the list of shared handlers for the interrupt. That's why the dev_id pointer must be unique.

A driver using a shared handler needs to be careful about one more thing: it can't play with enable_irq or disable_irq. If it does, things might go haywire for other devices sharing the line; disabling another device's interrupts for even a short time may create latencies that are problematic for that device and it's user. Generally, the programmer must remember that his driver doesn't own the IRQ, and its behavior should be more "social" than is necessary if one owns the interrupt line.

10.5.2. Running the Handler

10.6. Interrupt-Driven I/O

Whenever a data transfer to or from the managed hardware might be delayed for any reason, the driver writer should implement buffering. Data buffers help to detach data transmission and reception from the write and read system calls, and overall system performance benefits.

A good buffering mechanism leads to interrupt-driven I/O, in which an input buffer is filled at interrupt time and is emptied by processes that read the device; an output buffer is filled by processes that write to the device and is emptied at interrupt time. An example of interrupt-driven output is the implementation of */dev/shortprint*.

For interrupt-driven data transfer to happen successfully, the hardware should be able to generate interrupts with the following semantics:

- For input, the device interrupts the processor when new data has arrived and is ready to be retrieved by the system processor. The actual actions to perform depend on whether the device uses I/O ports, memory mapping, or DMA.
- For output, the device delivers an interrupt either when it is ready to accept new data or to acknowledge a successful data transfer. Memory-mapped and DMA-capable devices usually generate interrupts to tell the system they are done with the buffer.

The timing relationships between a read or write and the actual arrival of data were introduced in Section 6.2.3 in Chapter 6.

10.6.1. A Write-Buffering Example

We have mentioned the shortprint driver a couple of times; now it is time to actually take a look. This module implements a very simple, output-oriented driver for the parallel port; it is sufficient, however, to enable the printing of files. If you chose to test this driver out, however, remember that you must pass the printer a file in a format it understands; not all printers respond well when given a stream of arbitrary data.

The shortprint driver maintains a one-page circular output buffer. When a user-space process writes data to the device, that data is fed into the buffer, but the write method does not actually perform any I/O. Instead, the core of shortp_write looks like this:

```
    while (written < count) {

        /* Hang out until some buffer space is available. */

        space = shortp_out_space(  );

        if (space <= 0) {

            if (wait_event_interruptible(shortp_out_queue,

                        (space = shortp_out_space(  )) > 0))

                goto out;

        }



        /* Move data into the buffer. */

        if ((space + written) > count)

            space = count - written;

        if (copy_from_user((char *) shortp_out_head, buf, space)) {
```

10.7. Quick Reference

These symbols related to interrupt management were introduced in this chapter:

```
#include <linux/interrupt.h>

int request_irq(unsigned int irq, irqreturn_t (*handler)( ), unsigned long

flags, const char *dev_name, void *dev_id);

void free_irq(unsigned int irq, void *dev_id);
```

Calls that register and unregister an interrupt handler.

```
#include <linux/irq.h.h>

int can_request_irq(unsigned int irq, unsigned long flags);
```

This function, available on the i386 and x86_64 architectures, returns a nonzero value if an attempt to allocate the given interrupt line succeeds.

```
#include <asm/signal.h>

SA_INTERRUPT

SA_SHIRQ

SA_SAMPLE_RANDOM
```

Flags for request_irq. SA_INTERRUPT requests installation of a fast handler (as opposed to a slow one). SA_SHIRQ installs a shared handler, and the third flag asserts that interrupt timestamps can be used to generate system entropy.

```
/proc/interrupts

/proc/stat
```

Filesystem nodes that report information about hardware interrupts and installed handlers.

```
unsigned long probe_irq_on(void);

int probe_irq_off(unsigned long);
```

Functions used by the driver when it has to probe to determine which interrupt line is being used by a device. The result of probe_irq_on must be passed back to probe_irq_off after the interrupt has been generated. The return value of probe_irq_off is the detected interrupt number.

Chapter 11. Data Types in the Kernel

Before we go on to more advanced topics, we need to stop for a quick note on portability issues. Modern versions of the Linux kernel are highly portable, running on numerous different architectures. Given the multiplatform nature of Linux, drivers intended for serious use should be portable as well.

But a core issue with kernel code is being able both to access data items of known length (for example, filesystem data structures or registers on device boards) and to exploit the capabilities of different processors (32-bit and 64-bit architectures, and possibly 16 bit as well).

Several of the problems encountered by kernel developers while porting x86 code to new architectures have been related to incorrect data typing. Adherence to strict data typing and compiling with the -Wall -Wstrict-prototypes flags can prevent most bugs.

Data types used by kernel data are divided into three main classes: standard C types such as int, explicitly sized types such as u32, and types used for specific kernel objects, such as pid_t. We are going to see when and how each of the three typing classes should be used. The final sections of the chapter talk about some other typical problems you might run into when porting driver code from the x86 to other platforms, and introduce the generalized support for linked lists exported by recent kernel headers.

If you follow the guidelines we provide, your driver should compile and run even on platforms on which you are unable to test it.

11.1. Use of Standard C Types

Although most programmers are accustomed to freely using standard types like int and long, writing device drivers requires some care to avoid typing conflicts and obscure bugs.

The problem is that you can't use the standard types when you need "a 2-byte filler" or "something representing a 4-byte string," because the normal C data types are not the same size on all architectures. To show the data size of the various C types, the datasize program has been included in the sample files provided on O'Reilly's FTP site in the directory *misc-progs*. This is a sample run of the program on an i386 system (the last four types shown are introduced in the next section):

```
morgana% misc-progs/datasize

arch   Size:  char  short  int  long   ptr long-long  u8 u16 u32 u64

i686           1     2     4     4     4     8        1   2   4   8
```

The program can be used to show that long integers and pointers feature a different size on 64-bit platforms, as demonstrated by running the program on different Linux computers:

```
 arch   Size:  char  short  int  long   ptr long-long  u8 u16 u32 u64

i386           1     2     4     4     4     8        1   2   4   8

alpha          1     2     4     8     8     8        1   2   4   8

armv4l         1     2     4     4     4     8        1   2   4   8

ia64           1     2     4     8     8     8        1   2   4   8

m68k           1     2     4     4     4     8        1   2   4   8

mips           1     2     4     4     4     8        1   2   4   8

ppc            1     2     4     4     4     8        1   2   4   8

sparc          1     2     4     4     4     8        1   2   4   8

sparc64        1     2     4     4     4     8        1   2   4   8

x86_64         1     2     4     8     8     8        1   2   4   8
```

It's interesting to note that the SPARC 64 architecture runs with a 32-bit user space, so pointers are 32 bits wide there, even though they are 64 bits wide in kernel space. This can be verified by loading the kdatasize module (available in the directory *misc-modules* within the sample files). The module reports size information at load time using printk and returns an error (so there's no need to unload it):

```
 kernel: arch   Size:  char short int long  ptr long-long u8 u16 u32 u64

kernel: sparc64         1    2    4    8    8     8       1   2   4   8
```

Although you must be careful when mixing different data types, sometimes there are good reasons to do so. One such situation is for memory addresses, which are special as far as the kernel is concerned. Although, conceptually, addresses are pointers, memory administration is often better accomplished by using an unsigned integer type; the kernel treats physical memory like a huge array, and a memory address is just an index into the array. Furthermore, a pointer is easily dereferenced; when dealing directly with memory addresses, you almost never want to dereference them in this manner. Using an integer type prevents this dereferencing, thus avoiding bugs. Therefore, generic memory addresses in the kernel are usually unsigned long, exploiting the fact that pointers and long integers are always the same size, at least on all the platforms currently supported by Linux.

For what it's worth, the C99 standard defines the intptr_t and uintptr_t types for an integer variable that can hold a pointer value. These types are almost unused in the 2.6 kernel, however.

11.2. Assigning an Explicit Size to Data Items

Sometimes kernel code requires data items of a specific size, perhaps to match predefined binary structures,[1] to communicate with user space, or to align data within structures by inserting "padding" fields (but refer to the Section 11.4.4 for information about alignment issues).

[1] This happens when reading partition tables, when executing a binary file, or when decoding a network packet.

The kernel offers the following data types to use whenever you need to know the size of your data. All the types are declared in <*asm/types.h*>, which, in turn, is included by <*linux/types.h*>:

```
u8;   /* unsigned byte (8 bits) */

u16;  /* unsigned word (16 bits) */

u32;  /* unsigned 32-bit value */

u64;  /* unsigned 64-bit value */
```

The corresponding signed types exist, but are rarely needed; just replace u with s in the name if you need them.

If a user-space program needs to use these types, it can prefix the names with a double underscore: _ _u8 and the other types are defined independent of _ _KERNEL_ _. If, for example, a driver needs to exchange binary structures with a program running in user space by means of ioctl, the header files should declare 32-bit fields in the structures as _ _u32.

It's important to remember that these types are Linux specific, and using them hinders porting software to other Unix flavors. Systems with recent compilers support the C99-standard types, such as uint8_t and uint32_t; if portability is a concern, those types can be used in favor of the Linux-specific variety.

You might also note that sometimes the kernel uses conventional types, such as unsigned int, for items whose dimension is architecture independent. This is usually done for backward compatibility. When u32 and friends were introduced in Version 1.1.67, the developers couldn't change existing data structures to the new types because the compiler issues a warning when there is a type mismatch between the structure field and the value being assigned to it.[2] Linus didn't expect the operating system (OS) he wrote for his own use to become multiplatform; as a result, old structures are sometimes loosely typed.

[2] As a matter of fact, the compiler signals type inconsistencies even if the two types are just different names for the same object, such as unsigned long and u32 on the PC.

11.3. Interface-Specific Types

Some of the commonly used data types in the kernel have their own typedef statements, thus preventing any portability problems. For example, a process identifier (pid) is usually pid_t instead of int. Using pid_t masks any possible difference in the actual data typing. We use the expression interface-specific to refer to a type defined by a library in order to provide an interface to a specific data structure.

Note that, in recent times, relatively few new interface-specific types have been defined. Use of the typedef statement has gone out of favor among many kernel developers, who would rather see the real type information used directly in the code, rather than hidden behind a user-defined type. Many older interface-specific types remain in the kernel, however, and they will not be going away anytime soon.

Even when no interface-specific type is defined, it's always important to use the proper data type in a way consistent with the rest of the kernel. A jiffy count, for instance, is always unsigned long, independent of its actual size, so the unsigned long type should always be used when working with jiffies. In this section we concentrate on use of _t types.

Many _t types are defined in *<linux/types.h>*, but the list is rarely useful. When you need a specific type, you'll find it in the prototype of the functions you need to call or in the data structures you use.

Whenever your driver uses functions that require such "custom" types and you don't follow the convention, the compiler issues a warning; if you use the -Wall compiler flag and are careful to remove all the warnings, you can feel confident that your code is portable.

The main problem with _t data items is that when you need to print them, it's not always easy to choose the right printk or printf format, and warnings you resolve on one architecture reappear on another. For example, how would you print a size_t, that is unsigned long on some platforms and unsigned int on some others?

Whenever you need to print some interface-specific data, the best way to do it is by casting the value to the biggest possible type (usually long or unsigned long) and then printing it through the corresponding format. This kind of tweaking won't generate errors or warnings because the format matches the type, and you won't lose data bits because the cast is either a null operation or an extension of the item to a bigger data type.

In practice, the data items we're talking about aren't usually meant to be printed, so the issue applies only to debugging messages. Most often, the code needs only to store and compare the interface-specific types, in addition to passing them as arguments to library or kernel functions.

Although _t types are the correct solution for most situations, sometimes the right type doesn't exist. This happens for some old interfaces that haven't yet been cleaned up.

The one ambiguous point we've found in the kernel headers is data typing for I/O functions, which is loosely defined (see the Section 9.2.6 in Chapter 9). The loose typing is mainly there for historical reasons, but it can create problems when writing code. For example, one can get into trouble by swapping the arguments to functions like outb; if there were a port_t type, the compiler would find this type of error.

11.4. Other Portability Issues

In addition to data typing, there are a few other software issues to keep in mind when writing a driver if you want it to be portable across Linux platforms.

A general rule is to be suspicious of explicit constant values. Usually the code has been parameterized using preprocessor macros. This section lists the most important portability problems. Whenever you encounter other values that have been parameterized, you can find hints in the header files and in the device drivers distributed with the official kernel.

11.4.1. Time Intervals

When dealing with time intervals, don't assume that there are 1000 jiffies per second. Although this is currently true for the i386 architecture, not every Linux platform runs at this speed. The assumption can be false even for the x86 if you play with the HZ value (as some people do), and nobody knows what will happen in future kernels. Whenever you calculate time intervals using jiffies, scale your times using HZ (the number of timer interrupts per second). For example, to check against a timeout of half a second, compare the elapsed time against HZ/2. More generally, the number of jiffies corresponding to msec milliseconds is always msec*HZ/1000.

11.4.2. Page Size

When playing games with memory, remember that a memory page is PAGE_SIZE bytes, not 4 KB. Assuming that the page size is 4 KB and hardcoding the value is a common error among PC programmers, instead, supported platforms show page sizes from 4 KB to 64 KB, and sometimes they differ between different implementations of the same platform. The relevant macros are PAGE_SIZE and PAGE_SHIFT. The latter contains the number of bits to shift an address to get its page number. The number currently is 12 or greater for pages that are 4 KB and larger. The macros are defined in *<asm/page.h>*; user-space programs can use the getpagesize library function if they ever need the information.

Let's look at a nontrivial situation. If a driver needs 16 KB for temporary data, it shouldn't specify an order of 2 to get_free_pages. You need a portable solution. Such a solution, fortunately, has been written by the kernel developers and is called get_order:

```
 #include <asm/page.h>

int order = get_order(16*1024);

buf = get_free_pages(GFP_KERNEL, order);
```

Remember that the argument to get_order must be a power of two.

11.4.3. Byte Order

Be careful not to make assumptions about byte ordering. Whereas the PC stores multibyte values low-byte first (little end first, thus little-endian), some high-level platforms work the other way (big-endian). Whenever possible, your code should be written such that it does not care about byte ordering in the data it manipulates. However, sometimes a driver needs to build an integer number out of single bytes or do the opposite, or it must communicate with a device that expects a specific order.

The include file *<asm/byteorder.h>* defines either _ _BIG_ENDIAN or _ _LITTLE_ENDIAN, depending on the processor's byte ordering. When dealing with byte ordering issues, you could code a bunch of #ifdef _ _LITTLE_ENDIAN conditionals, but there is a better way. The Linux kernel defines a set of macros that handle conversions between the processor's byte ordering and that of the data you need to store or load in a specific byte order. For example:

```
 u32 cpu_to_le32 (u32);

u32 le32_to_cpu (u32);
```

These two macros convert a value from whatever the CPU uses to an unsigned, little-endian, 32-bit quantity and back. They work whether your CPU is big-endian or little-endian and, for that matter, whether it is a 32-bit

11.5. Linked Lists

Operating system kernels, like many other programs, often need to maintain lists of data structures. The Linux kernel has, at times, been host to several linked list implementations at the same time. To reduce the amount of duplicated code, the kernel developers have created a standard implementation of circular, doubly linked lists; others needing to manipulate lists are encouraged to use this facility.

When working with the linked list interface, you should always bear in mind that the list functions perform no locking. If there is a possibility that your driver could attempt to perform concurrent operations on the same list, it is your responsibility to implement a locking scheme. The alternatives (corrupted list structures, data loss, kernel panics) tend to be difficult to diagnose.

To use the list mechanism, your driver must include the file *<linux/list.h>*. This file defines a simple structure of type list_head:

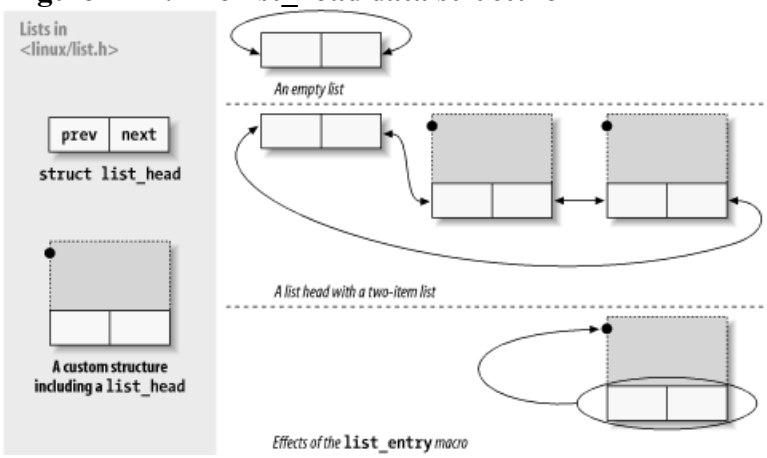
```
struct list_head {

    struct list_head *next, *prev;

};
```

Linked lists used in real code are almost invariably made up of some type of structure, each one describing one entry in the list. To use the Linux list facility in your code, you need only embed a list_head inside the structures that make up the list. If your driver maintains a list of things to do, say, its declaration would look something like this:

```
struct todo_struct {

    struct list_head list;

    int priority; /* driver specific */

    /* ... add other driver-specific fields */

};
```

The head of the list is usually a standalone list_head structure. Figure 11-1 shows how the simple struct list_head is used to maintain a list of data structures.

Figure 11-1. The list_head data structure



List heads must be initialized prior to use with the INIT_LIST_HEAD macro. A "things to do" list head could be declared and initialized with:

```
struct list_head todo_list;



INIT_LIST_HEAD(&todo_list);
```

11.6. Quick Reference

The following symbols were introduced in this chapter:

#include <linux/types.h>

typedef u8;

typedef u16;

typedef u32;

typedef u64;

Types guaranteed to be 8-, 16-, 32-, and 64-bit unsigned integer values. The equivalent signed types exist as well. In user space, you can refer to the types as _ _u8, _ _u16, and so forth.

#include <asm/page.h>

PAGE_SIZE

PAGE_SHIFT

Symbols that define the number of bytes per page for the current architecture and the number of bits in the page offset (12 for 4-KB pages and 13 for 8-KB pages).

#include <asm/byteorder.h>

_ _LITTLE_ENDIAN

_ _BIG_ENDIAN

Only one of the two symbols is defined, depending on the architecture.

#include <asm/byteorder.h>

u32 _ _cpu_to_le32 (u32);

u32 _ _le32_to_cpu (u32);

Functions that convert between known byte orders and that of the processor. There are more than 60 such functions; see the various files in *include/linux/byteorder/* for a full list and the ways in which they are defined.

#include <asm/unaligned.h>

Chapter 12. PCI Drivers

While Chapter 9 introduced the lowest levels of hardware control, this chapter provides an overview of the higher-level bus architectures. A bus is made up of both an electrical interface and a programming interface. In this chapter, we deal with the programming interface.

This chapter covers a number of bus architectures. However, the primary focus is on the kernel functions that access Peripheral Component Interconnect (PCI) peripherals, because these days the PCI bus is the most commonly used peripheral bus on desktops and bigger computers. The bus is the one that is best supported by the kernel. ISA is still common for electronic hobbyists and is described later, although it is pretty much a bare-metal kind of bus, and there isn't much to say in addition to what is covered in Chapter 9 and Chapter 10.

12.1. The PCI Interface

Although many computer users think of PCI as a way of laying out electrical wires, it is actually a complete set of specifications defining how different parts of a computer should interact.

The PCI specification covers most issues related to computer interfaces. We are not going to cover it all here; in this section, we are mainly concerned with how a PCI driver can find its hardware and gain access to it. The probing techniques discussed in Chapter 12 and Chapter 10 can be used with PCI devices, but the specification offers an alternative that is preferable to probing.

The PCI architecture was designed as a replacement for the ISA standard, with three main goals: to get better performance when transferring data between the computer and its peripherals, to be as platform independent as possible, and to simplify adding and removing peripherals to the system.

The PCI bus achieves better performance by using a higher clock rate than ISA; its clock runs at 25 or 33 MHz (its actual rate being a factor of the system clock), and 66-MHz and even 133-MHz implementations have recently been deployed as well. Moreover, it is equipped with a 32-bit data bus, and a 64-bit extension has been included in the specification. Platform independence is often a goal in the design of a computer bus, and it's an especially important feature of PCI, because the PC world has always been dominated by processor-specific interface standards. PCI is currently used extensively on IA-32, Alpha, PowerPC, SPARC64, and IA-64 systems, and some other platforms as well.

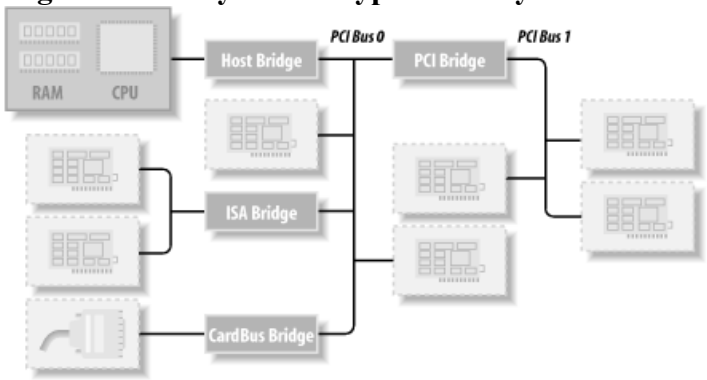
What is most relevant to the driver writer, however, is PCI's support for autodetection of interface boards. PCI devices are jumperless (unlike most older peripherals) and are automatically configured at boot time. Then, the device driver must be able to access configuration information in the device in order to complete initialization. This happens without the need to perform any probing.

12.1.1. PCI Addressing

Each PCI peripheral is identified by a *bus* number, a *device* number, and a *function* number. The PCI specification permits a single system to host up to 256 buses, but because 256 buses are not sufficient for many large systems, Linux now supports PCI *domains*. Each PCI domain can host up to 256 buses. Each bus hosts up to 32 devices, and each device can be a multifunction board (such as an audio device with an accompanying CD-ROM drive) with a maximum of eight functions. Therefore, each function can be identified at hardware level by a 16-bit address, or key. Device drivers written for Linux, though, don't need to deal with those binary addresses, because they use a specific data structure, called pci_dev, to act on the devices.

Most recent workstations feature at least two PCI buses. Plugging more than one bus in a single system is accomplished by means of bridges, special-purpose PCI peripherals whose task is joining two buses. The overall layout of a PCI system is a tree where each bus is connected to an upper-layer bus, up to bus 0 at the root of the tree. The CardBus PC-card system is also connected to the PCI system via bridges. A typical PCI system is represented in Figure 12-1, where the various bridges are highlighted.

Figure 12-1. Layout of a typical PCI system



The 16-bit hardware addresses associated with PCI peripherals, although mostly hidden in the struct pci_dev object, are still visible occasionally, especially when lists of devices are being used. One such situation is the

12.2. A Look Back: ISA

The ISA bus is quite old in design and is a notoriously poor performer, but it still holds a good part of the market for extension devices. If speed is not important and you want to support old motherboards, an ISA implementation is preferable to PCI. An additional advantage of this old standard is that if you are an electronic hobbyist, you can easily build your own ISA devices, something definitely not possible with PCI.

On the other hand, a great disadvantage of ISA is that it's tightly bound to the PC architecture; the interface bus has all the limitations of the 80286 processor and causes endless pain to system programmers. The other great problem with the ISA design (inherited from the original IBM PC) is the lack of geographical addressing, which has led to many problems and lengthy unplug-rejumper-plug-test cycles to add new devices. It's interesting to note that even the oldest Apple II computers were already exploiting geographical addressing, and they featured jumperless expansion boards.

Despite its great disadvantages, ISA is still used in several unexpected places. For example, the VR41xx series of MIPS processors used in several palmtops features an ISA-compatible expansion bus, strange as it seems. The reason behind these unexpected uses of ISA is the extreme low cost of some legacy hardware, such as 8390-based Ethernet cards, so a CPU with ISA electrical signaling can easily exploit the awful, but cheap, PC devices.

12.2.1. Hardware Resources

An ISA device can be equipped with I/O ports, memory areas, and interrupt lines.

Even though the x86 processors support 64 KB of I/O port memory (i.e., the processor asserts 16 address lines), some old PC hardware decodes only the lowest 10 address lines. This limits the usable address space to 1024 ports, because any address in the range 1 KB to 64 KB is mistaken for a low address by any device that decodes only the low address lines. Some peripherals circumvent this limitation by mapping only one port into the low kilobyte and using the high address lines to select between different device registers. For example, a device mapped at 0x340 can safely use port 0x740, 0xB40, and so on.

If the availability of I/O ports is limited, memory access is still worse. An ISA device can use only the memory range between 640 KB and 1 MB and between 15 MB and 16 MB for I/O register and device control. The 640-KB to 1-MB range is used by the PC BIOS, by VGA-compatible video boards, and by various other devices, leaving little space available for new devices. Memory at 15 MB, on the other hand, is not directly supported by Linux, and hacking the kernel to support it is a waste of programming time nowadays.

The third resource available to ISA device boards is interrupt lines. A limited number of interrupt lines is routed to the ISA bus, and they are shared by all the interface boards. As a result, if devices aren't properly configured, they can find themselves using the same interrupt lines.

Although the original ISA specification doesn't allow interrupt sharing across devices, most device boards allow it.[5] Interrupt sharing at the software level is described in Chapter 10.

[5] The problem with interrupt sharing is a matter of electrical engineering: if a device drives the signal line inactive—by applying a low-impedance voltage level—the interrupt can't be shared. If, on the other hand, the device uses a pull-up resistor to the inactive logic level, sharing is possible. This is the norm nowadays. However, there's still a potential risk of losing interrupt events since ISA interrupts are edge triggered instead of level triggered. Edge-triggered interrupts are easier to implement in hardware but don't lend themselves to safe sharing.

12.2.2. ISA Programming

As far as programming is concerned, there's no specific aid in the kernel or the BIOS to ease access to ISA devices (like there is, for example, for PCI). The only facilities you can use are the registries of I/O ports and IRQ lines, described in Section 10.2.

The programming techniques shown throughout the first part of this book apply to ISA devices; the driver can probe for I/O ports, and the interrupt line must be autodetected with one of the techniques shown in Section 10.2.2.

12.3. PC/104 and PC/104+

Currently in the industrial world, two bus architectures are quite fashionable: PC/104 and PC/104+. Both are standard in PC-class single-board computers.

Both standards refer to specific form factors for printed circuit boards, as well as electrical/mechanical specifications for board interconnections. The practical advantage of these buses is that they allow circuit boards to be stacked vertically using a plug-and-socket kind of connector on one side of the device.

The electrical and logical layout of the two buses is identical to ISA (PC/104) and PCI (PC/104+), so software won't notice any difference between the usual desktop buses and these two.

12.4. Other PC Buses

PCI and ISA are the most commonly used peripheral interfaces in the PC world, but they aren't the only ones. Here's a summary of the features of other buses found in the PC market.

12.4.1. MCA

Micro Channel Architecture (MCA) is an IBM standard used in PS/2 computers and some laptops. At the hardware level, Micro Channel has more features than ISA. It supports multimaster DMA, 32-bit address and data lines, shared interrupt lines, and geographical addressing to access per-board configuration registers. Such registers are called Programmable Option Select (POS), but they don't have all the features of the PCI registers. Linux support for Micro Channel includes functions that are exported to modules.

A device driver can read the integer value MCA_bus to see if it is running on a Micro Channel computer. If the symbol is a preprocessor macro, the macro MCA_bus_ _is_a_macro is defined as well. If MCA_bus_ _is_a_macro is undefined, then MCA_bus is an integer variable exported to modularized code. Both MCA_BUS and MCA_bus_ _is_a_macro are defined in *<asm/processor.h>*.

12.4.2. EISA

The Extended ISA (EISA) bus is a 32-bit extension to ISA, with a compatible interface connector; ISA device boards can be plugged into an EISA connector. The additional wires are routed under the ISA contacts.

Like PCI and MCA, the EISA bus is designed to host jumperless devices, and it has the same features as MCA: 32-bit address and data lines, multimaster DMA, and shared interrupt lines. EISA devices are configured by software, but they don't need any particular operating system support. EISA drivers already exist in the Linux kernel for Ethernet devices and SCSI controllers.

An EISA driver checks the value EISA_bus to determine if the host computer carries an EISA bus. Like MCA_bus, EISA_bus is either a macro or a variable, depending on whether EISA_bus_ _is_a_macro is defined. Both symbols are defined in *<asm/processor.h>*.

The kernel has full EISA support for devices with sysfs and resource management functionality. This is located in the drivers/eisa directory.

12.4.3. VLB

Another extension to ISA is the VESA Local Bus (VLB) interface bus, which extends the ISA connectors by adding a third lengthwise slot. A device can just plug into this extra connector (without plugging in the two associated ISA connectors), because the VLB slot duplicates all important signals from the ISA connectors. Such "standalone" VLB peripherals not using the ISA slot are rare, because most devices need to reach the back panel so that their external connectors are available.

The VESA bus is much more limited in its capabilities than the EISA, MCA, and PCI buses and is disappearing from the market. No special kernel support exists for VLB. However, both the Lance Ethernet driver and the IDE disk driver in Linux 2.0 can deal with VLB versions of their devices.

12.5. SBus

While most computers nowadays are equipped with a PCI or ISA interface bus, most older SPARC-based workstations use SBus to connect their peripherals.

SBus is quite an advanced design, although it has been around for a long time. It is meant to be processor independent (even though only SPARC computers use it) and is optimized for I/O peripheral boards. In other words, you can't plug additional RAM into SBus slots (RAM expansion boards have long been forgotten even in the ISA world, and PCI does not support them either). This optimization is meant to simplify the design of both hardware devices and system software, at the expense of some additional complexity in the motherboard.

This I/O bias of the bus results in peripherals using virtual addresses to transfer data, thus bypassing the need to allocate a contiguous DMA buffer. The motherboard is responsible for decoding the virtual addresses and mapping them to physical addresses. This requires attaching an MMU (memory management unit) to the bus; the chipset in charge of the task is called IOMMU. Although somehow more complex than using physical addresses on the interface bus, this design is greatly simplified by the fact that SPARC processors have always been designed by keeping the MMU core separate from the CPU core (either physically or at least conceptually). Actually, this design choice is shared by other smart processor designs and is beneficial overall. Another feature of this bus is that device boards exploit massive geographical addressing, so there's no need to implement an address decoder in every peripheral or to deal with address conflicts.

SBus peripherals use the Forth language in their PROMs to initialize themselves. Forth was chosen because the interpreter is lightweight and, therefore, can be easily implemented in the firmware of any computer system. In addition, the SBus specification outlines the boot process, so that compliant I/O devices fit easily into the system and are recognized at system boot. This was a great step to support multi-platform devices; it's a completely different world from the PC-centric ISA stuff we were used to. However, it didn't succeed for a variety of commercial reasons.

Although current kernel versions offer quite full-featured support for SBus devices, the bus is used so little nowadays that it's not worth covering in detail here. Interested readers can look at source files in *arch/sparc/kernel* and *arch/sparc/mm*.

12.6. NuBus

Another interesting, but nearly forgotten, interface bus is NuBus. It is found on older Mac computers (those with the M68k family of CPUs).

All of the bus is memory-mapped (like everything with the M68k), and the devices are only geographically addressed. This is good and typical of Apple, as the much older Apple II already had a similar bus layout. What is bad is that it's almost impossible to find documentation on NuBus, due to the close-everything policy Apple has always followed with its Mac computers (and unlike the previous Apple II, whose source code and schematics were available at little cost).

The file *drivers/nubus/nubus.c* includes almost everything we know about this bus, and it's interesting reading; it shows how much hard reverse engineering developers had to do.

12.7. External Buses

One of the most recent entries in the field of interface buses is the whole class of external buses. This includes USB, FireWire, and IEEE1284 (parallel-port-based external bus). These interfaces are somewhat similar to older and not-so-external technology, such as PCMCIA/CardBus and even SCSI.

Conceptually, these buses are neither full-featured interface buses (like PCI is) nor dumb communication channels (like the serial ports are). It's hard to classify the software that is needed to exploit their features, as it's usually split into two levels: the driver for the hardware controller (like drivers for PCI SCSI adaptors or PCI controllers introduced in the Section 12.1) and the driver for the specific "client" device (like sd.c handles generic SCSI disks and so-called PCI drivers deal with cards plugged in the bus).

12.8. Quick Reference

This section summarizes the symbols introduced in the chapter:

#include <linux/pci.h>

Header that includes symbolic names for the PCI registers and several vendor and device ID values.

struct pci_dev;

Structure that represents a PCI device within the kernel.

struct pci_driver;

Structure that represents a PCI driver. All PCI drivers must define this.

struct pci_device_id;

Structure that describes the types of PCI devices this driver supports.

int pci_register_driver(struct pci_driver *drv);

int pci_module_init(struct pci_driver *drv);

void pci_unregister_driver(struct pci_driver *drv);

Functions that register or unregister a PCI driver from the kernel.

struct pci_dev *pci_find_device(unsigned int vendor, unsigned int device,

struct pci_dev *from);

struct pci_dev *pci_find_device_reverse(unsigned int vendor, unsigned int

device, const struct pci_dev *from);

struct pci_dev *pci_find_subsys (unsigned int vendor, unsigned int device,

unsigned int ss_vendor, unsigned int ss_device, const struct pci_dev *from);

struct pci_dev *pci_find_class(unsigned int class, struct pci_dev *from);

Functions that search the device list for devices with a specific signature or those belonging to a specific class. The return value is NULL if none is found. from is used to continue a search; it must be NULL the first time you call either function, and it must point to the device just found if you are searching for more devices. These functions are not recommended to be used, use the pci_get_ variants instead.

Chapter 13. USB Drivers

The universal serial bus (USB) is a connection between a host computer and a number of peripheral devices. It was originally created to replace a wide range of slow and different buses—the parallel, serial, and keyboard connections—with a single bus type that all devices could connect to.[1] USB has grown beyond these slow connections and now supports almost every type of device that can be connected to a PC. The latest revision of the USB specification added high-speed connections with a theoretical speed limit of 480 MBps.

[1] Portions of this chapter are based on the in-kernel documentation for the Linux kernel USB code, which were written by the kernel USB developers and released under the GPL.

Topologically, a USB subsystem is not laid out as a bus; it is rather a tree built out of several point-to-point links. The links are four-wire cables (ground, power, and two signal wires) that connect a device and a hub, just like twisted-pair Ethernet. The USB host controller is in charge of asking every USB device if it has any data to send. Because of this topology, a USB device can never start sending data without first being asked to by the host controller. This configuration allows for a very easy plug-and-play type of system, whereby devices can be automatically configured by the host computer.

The bus is very simple at the technological level, as it's a single-master implementation in which the host computer polls the various peripheral devices. Despite this intrinsic limitation, the bus has some interesting features, such as the ability for a device to request a fixed bandwidth for its data transfers in order to reliably support video and audio I/O. Another important feature of USB is that it acts merely as a communication channel between the device and the host, without requiring specific meaning or structure to the data it delivers.[2]

[2] Actually, some structure is there, but it mostly reduces to a requirement for the communication to fit into one of a few predefined classes: a keyboard won't allocate bandwidth, for example, while some video cameras will.

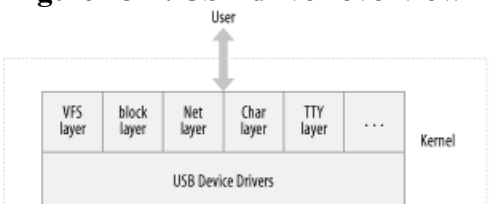
The USB protocol specifications define a set of standards that any device of a specific type can follow. If a device follows that standard, then a special driver for that device is not necessary. These different types are called classes and consist of things like storage devices, keyboards, mice, joysticks, network devices, and modems. Other types of devices that do not fit into these classes require a special vendor-specific driver to be written for that specific device. Video devices and USB-to-serial devices are a good example where there is no defined standard, and a driver is needed for every different device from different manufacturers.

These features, together with the inherent hotplug capability of the design, make USB a handy, low-cost mechanism to connect (and disconnect) several devices to the computer without the need to shut the system down, open the cover, and swear over screws and wires.

The Linux kernel supports two main types of USB drivers: drivers on a host system and drivers on a device. The USB drivers for a host system control the USB devices that are plugged into it, from the host's point of view (a common USB host is a desktop computer.) The USB drivers in a device, control how that single device looks to the host computer as a USB device. As the term "USB device drivers" is very confusing, the USB developers have created the term "USB gadget drivers" to describe the drivers that control a USB device that connects to a computer (remember that Linux also runs in those tiny embedded devices, too.) This chapter details how the USB system that runs on a desktop computer works. USB gadget drivers are outside the realm of this book at this point in time.

As Chapter 13 shows, USB drivers live between the different kernel subsytems (block, net, char, etc.) and the USB hardware controllers. The USB core provides an interface for USB drivers to use to access and control the USB hardware, without having to worry about the different types of USB hardware controllers that are present on the system.

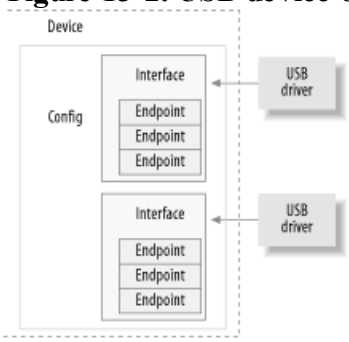
Figure 13-1. USB driver overview



13.1. USB Device Basics

A USB device is a very complex thing, as described in the official USB documentation (available at http://www.usb.org). Fortunately, the Linux kernel provides a subsystem called the *USB core* to handle most of the complexity. This chapter describes the interaction between a driver and the USB core. Figure 13-1 shows how USB devices consist of configurations, interfaces, and endpoints and how USB drivers bind to USB interfaces, not the entire USB device.

Figure 13-2. USB device overview



13.1.1. Endpoints

The most basic form of USB communication is through something called an *endpoint*. A USB endpoint can carry data in only one direction, either from the host computer to the device (called an OUT endpoint) or from the device to the host computer (called an IN endpoint). Endpoints can be thought of as unidirectional pipes.

A USB endpoint can be one of four different types that describe how the data is transmitted:

CONTROL

Control endpoints are used to allow access to different parts of the USB device. They are commonly used for configuring the device, retrieving information about the device, sending commands to the device, or retrieving status reports about the device. These endpoints are usually small in size. Every USB device has a control endpoint called "endpoint 0" that is used by the USB core to configure the device at insertion time. These transfers are guaranteed by the USB protocol to always have enough reserved bandwidth to make it through to the device.

INTERRUPT

Interrupt endpoints transfer small amounts of data at a fixed rate every time the USB host asks the device for data. These endpoints are the primary transport method for USB keyboards and mice. They are also commonly used to send data to USB devices to control the device, but are not generally used to transfer large amounts of data. These transfers are guaranteed by the USB protocol to always have enough reserved bandwidth to make it through.

BULK

Bulk endpoints transfer large amounts of data. These endpoints are usually much larger (they can hold more characters at once) than interrupt endpoints. They are common for devices that need to transfer any data that must get through with no data loss. These transfers are not guaranteed by the USB protocol to always make it through in a specific amount of time. If there is not enough room on the bus to send the whole BULK packet, it is split up across multiple transfers to or from the device. These endpoints are common on printers, storage, and network devices.

ISOCHRONOUS

13.2. USB and Sysfs

Due to the complexity of a single USB physical device, the representation of that device in sysfs is also quite complex. Both the physical USB device (as represented by a struct usb_device) and the individual USB interfaces (as represented by a struct usb_interface) are shown in sysfs as individual devices. (This is because both of those structures contain a struct device structure.) As an example, for a simple USB mouse that contains only one USB interface, the following would be the sysfs directory tree for that device:

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1

|-- 2-1:1.0

|   |-- bAlternateSetting

|   |-- bInterfaceClass

|   |-- bInterfaceNumber

|   |-- bInterfaceProtocol

|   |-- bInterfaceSubClass

|   |-- bNumEndpoints

|   |-- detach_state

|   |-- iInterface

|   `-- power

|       `-- state

|-- bConfigurationValue

|-- bDeviceClass

|-- bDeviceProtocol

|-- bDeviceSubClass

|-- bMaxPower

|-- bNumConfigurations

|-- bNumInterfaces

|-- bcdDevice

|-- bmAttributes

|-- detach_state

|-- devnum

|-- idProduct

|-- idVendor

|-- maxchild

|-- power

|   `-- state

|-- speed

`-- version
```

13.3. USB Urbs

The USB code in the Linux kernel communicates with all USB devices using something called a urb (USB request block). This request block is described with the struct urb structure and can be found in the include/linux/usb.h file.

A urb is used to send or receive data to or from a specific USB endpoint on a specific USB device in an asynchronous manner. It is used much like a kiocb structure is used in the filesystem async I/O code or as a struct skbuff is used in the networking code. A USB device driver may allocate many urbs for a single endpoint or may reuse a single urb for many different endpoints, depending on the need of the driver. Every endpoint in a device can handle a queue of urbs, so that multiple urbs can be sent to the same endpoint before the queue is empty. The typical lifecycle of a urb is as follows:

- Created by a USB device driver.
- Assigned to a specific endpoint of a specific USB device.
- Submitted to the USB core, by the USB device driver.
- Submitted to the specific USB host controller driver for the specified device by the USB core.
- Processed by the USB host controller driver that makes a USB transfer to the device.
- When the urb is completed, the USB host controller driver notifies the USB device driver.

Urbs can also be canceled any time by the driver that submitted the urb, or by the USB core if the device is removed from the system. urbs are dynamically created and contain an internal reference count that enables them to be automatically freed when the last user of the urb releases it.

The procedure described in this chapter for handling urbs is useful, because it permits streaming and other complex, overlapping communications that allow drivers to achieve the highest possible data transfer speeds. But less cumbersome procedures are available if you just want to send individual bulk or control messages and do not care about data throughput rates. (See the Section 13.5.)

13.3.1. struct urb

The fields of the struct urb structure that matter to a USB device driver are:

struct usb_device *dev

Pointer to the struct usb_device to which this urb is sent. This variable must be initialized by the USB driver before the urb can be sent to the USB core.

unsigned int pipe

Endpoint information for the specific struct usb_device that this urb is to be sent to. This variable must be initialized by the USB driver before the urb can be sent to the USB core.

To set fields of this structure, the driver uses the following functions as appropriate, depending on the direction of traffic. Note that every endpoint can be of only one type.

13.4. Writing a USB Driver

The approach to writing a USB device driver is similar to a pci_driver: the driver registers its driver object with the USB subsystem and later uses vendor and device identifiers to tell if its hardware has been installed.

13.4.1. What Devices Does the Driver Support?

The struct usb_device_id structure provides a list of different types of USB devices that this driver supports. This list is used by the USB core to decide which driver to give a device to, and by the hotplug scripts to decide which driver to automatically load when a specific device is plugged into the system.

The struct usb_device_id structure is defined with the following fields:

__u16 match_flags

Determines which of the following fields in the structure the device should be matched against. This is a bit field defined by the different USB_DEVICE_ID_MATCH_* values specified in the *include/linux/mod_devicetable.h* file. This field is usually never set directly but is initialized by the USB_DEVICE type macros described later.

__u16 idVendor

The USB vendor ID for the device. This number is assigned by the USB forum to its members and cannot be made up by anyone else.

__u16 idProduct

The USB product ID for the device. All vendors that have a vendor ID assigned to them can manage their product IDs however they choose to.

__u16 bcdDevice_lo

__u16 bcdDevice_hi

Define the low and high ends of the range of the vendor-assigned product version number. The bcdDevice_hi value is inclusive; its value is the number of the highest-numbered device. Both of these values are expressed in binary-coded decimal (BCD) form. These variables, combined with the idVendor and idProduct, are used to define a specific version of a device.

__u8 bDeviceClass

__u8 bDeviceSubClass

__u8 bDeviceProtocol

Define the class, subclass, and protocol of the device, respectively. These numbers are assigned by the USB forum and are defined in the USB specification. These values specify the behavior for the whole device, including all interfaces on this device.

__u8 bInterfaceClass

13.5. USB Transfers Without Urbs

Sometimes a USB driver does not want to go through all of the hassle of creating a struct urb, initializing it, and then waiting for the urb completion function to run, just to send or receive some simple USB data. Two functions are available to provide a simpler interface.

13.5.1. usb_bulk_msg

usb_bulk_msg creates a USB bulk urb and sends it to the specified device, then waits for it to complete before returning to the caller. It is defined as:

```
int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe,

                 void *data, int len, int *actual_length,

                 int timeout);
```

The parameters of this function are:

struct usb_device *usb_dev

A pointer to the USB device to send the bulk message to.

unsigned int pipe

The specific endpoint of the USB device to which this bulk message is to be sent. This value is created with a call to either usb_sndbulkpipe or usb_rcvbulkpipe.

void *data

A pointer to the data to send to the device if this is an OUT endpoint. If this is an IN endpoint, this is a pointer to where the data should be placed after being read from the device.

int len

The length of the buffer that is pointed to by the data parameter.

int *actual_length

A pointer to where the function places the actual number of bytes that have either been transferred to the device or received from the device, depending on the direction of the endpoint.

int timeout

The amount of time, in jiffies, that should be waited before timing out. If this value is 0, the function waits forever for the message to complete.

If the function is successful, the return value is 0; otherwise, a negative error number is returned. This error number matches up with the error numbers previously described for urbs in Section 13.3.1. If successful, the actual_length parameter contains the number of bytes that were transferred or received from this message.

The following is an example of using this function call:

```
/* do a blocking bulk read to get data from the device */

retval = usb_bulk_msg(dev->udev,
```

13.6. Quick Reference

This section summarizes the symbols introduced in the chapter:

#include <linux/usb.h>

Header file where everything related to USB resides. It must be included by all USB device drivers.

struct usb_driver;

Structure that describes a USB driver.

struct usb_device_id;

Structure that describes the types of USB devices this driver supports.

int usb_register(struct usb_driver *d);

void usb_deregister(struct usb_driver *d);

Functions used to register and unregister a USB driver from the USB core.

struct usb_device *interface_to_usbdev(struct usb_interface *intf);

Retrieves the controlling struct usb_device * out of a struct usb_interface *.

struct usb_device;

Structure that controls an entire USB device.

struct usb_interface;

Main USB device structure that all USB drivers use to communicate with the USB core.

void usb_set_intfdata(struct usb_interface *intf, void *data);

void *usb_get_intfdata(struct usb_interface *intf);

Functions to set and get access to the private data pointer section within the struct usb_interface.

struct usb_class_driver;

A structure that describes a USB driver that wants to use the USB major number to communicate with user-space programs.

int usb_register_dev(struct usb_interface *intf, struct usb_class_driver

Chapter 14. The Linux Device Model

One of the stated goals for the 2.5 development cycle was the creation of a unified device model for the kernel. Previous kernels had no single data structure to which they could turn to obtain information about how the system is put together. Despite this lack of information, things worked well for some time. The demands of newer systems, with their more complicated topologies and need to support features such as power management, made it clear, however, that a general abstraction describing the structure of the system was needed.

The 2.6 device model provides that abstraction. It is now used within the kernel to support a wide variety of tasks, including:

Power management and system shutdown

These require an understanding of the system's structure. For example, a USB host adaptor cannot be shut down before dealing with all of the devices connected to that adaptor. The device model enables a traversal of the system's hardware in the right order.

Communications with user space

The implementation of the sysfs virtual filesystem is tightly tied into the device model and exposes the structure represented by it. The provision of information about the system to user space and knobs for changing operating parameters is increasingly done through sysfs and, therefore, through the device model.

Hotpluggable devices

Computer hardware is increasingly dynamic; peripherals can come and go at the whim of the user. The hotplug mechanism used within the kernel to handle and (especially) communicate with user space about the plugging and unplugging of devices is managed through the device model.

Device classes

Many parts of the system have little interest in how devices are connected, but they need to know what kinds of devices are available. The device model includes a mechanism for assigning devices to *classes*, which describe those devices at a higher, functional level and allow them to be discovered from user space.

Object lifecycles

Many of the functions described above, including hotplug support and sysfs, complicate the creation and manipulation of objects created within the kernel. The implementation of the device model required the creation of a set of mechanisms for dealing with object lifecycles, their relationships to each other, and their representation in user space.

The Linux device model is a complex data structure. For example, consider Chapter 14, which shows (in simplified form) a tiny piece of the device model structure associated with a USB mouse. Down the center of the diagram, we see the part of the core "devices" tree that shows how the mouse is connected to the system. The "bus" tree tracks what is connected to each bus, while the subtree under "classes" concerns itself with the functions provided by the devices, regardless of how they are connected. The device model tree on even a simple system contains hundreds of nodes like those shown in the diagram; it is a difficult data structure to visualize as a whole.

Figure 14-1. A small piece of the device model

14.1. Kobjects, Ksets, and Subsystems

The *kobject* is the fundamental structure that holds the device model together. It was initially conceived as a simple reference counter, but its responsibilities have grown over time, and so have its fields. The tasks handled by struct kobject and its supporting code now include:

Reference counting of objects

Often, when a kernel object is created, there is no way to know just how long it will exist. One way of tracking the lifecycle of such objects is through reference counting. When no code in the kernel holds a reference to a given object, that object has finished its useful life and can be deleted.

Sysfs representation

Every object that shows up in sysfs has, underneath it, a kobject that interacts with the kernel to create its visible representation.

Data structure glue

The device model is, in its entirety, a fiendishly complicated data structure made up of multiple hierarchies with numerous links between them. The kobject implements this structure and holds it together.

Hotplug event handling

The kobject subsystem handles the generation of events that notify user space about the comings and goings of hardware on the system.

One might conclude from the preceding list that the kobject is a complicated structure. One would be right. By looking at one piece at a time, however, it is possible to understand this structure and how it works.

14.1.1. Kobject Basics

A kobject has the type struct kobject; it is defined in <*linux/kobject.h*>. That file also includes declarations for a number of other structures related to kobjects and, of course, a long list of functions for manipulating them.

14.1.1.1 Embedding kobjects

Before we get into the details, it is worth taking a moment to understand how kobjects are used. If you look back at the list of functions handled by kobjects, you see that they are all services performed on behalf of other objects. A kobject, in other words, is of little interest on its own; it exists only to tie a higher-level object into the device model.

Thus, it is rare (even unknown) for kernel code to create a standalone kobject; instead, kobjects are used to control access to a larger, domain-specific object. To this end, kobjects are found embedded in other structures. If you are used to thinking of things in object-oriented terms, kobjects can be seen as a top-level, abstract class from which other classes are derived. A kobject implements a set of capabilities that are not particularly useful by themselves but that are nice to have in other objects. The C language does not allow for the direct expression of inheritance, so other techniques—such as embedding one structure in another—must be used.

As an example, let's look back at struct cdev, which we encountered in Chapter 3. That structure, as found in the 2.6.10 kernel, looks like this:

```
struct cdev {

    struct kobject kobj;

    struct module *owner;
```

14.2. Low-Level Sysfs Operations

Kobjects are the mechanism behind the sysfs virtual filesystem. For every directory found in sysfs, there is a kobject lurking somewhere within the kernel. Every kobject of interest also exports one or more *attributes*, which appear in that kobject's sysfs directory as files containing kernel-generated information. This section examines how kobjects and sysfs interact at a low level.

Code that works with sysfs should include <*linux/sysfs.h*>.

Getting a kobject to show up in sysfs is simply a matter of calling kobject_add. We have already seen that function as the way to add a kobject to a kset; creating entries in sysfs is also part of its job. There are a couple of things worth knowing about how the sysfs entry is created:

- Sysfs entries for kobjects are always directories, so a call to kobject_add results in the creation of a directory in sysfs. Usually that directory contains one or more attributes; we see how attributes are specified shortly.
- The name assigned to the kobject (with kobject_set_name) is the name used for the sysfs directory. Thus, kobjects that appear in the same part of the sysfs hierarchy must have unique names. Names assigned to kobjects should also be reasonable file names: they cannot contain the slash character, and the use of white space is strongly discouraged.
- The sysfs entry is located in the directory corresponding to the kobject's parent pointer. If parent is NULL when kobject_add is called, it is set to the kobject embedded in the new kobject's kset; thus, the sysfs hierarchy usually matches the internal hierarchy created with ksets. If both parent and kset are NULL, the sysfs directory is created at the top level, which is almost certainly not what you want.

Using the mechanisms we have described so far, we can use a kobject to create an empty directory in sysfs. Usually, you want to do something a little more interesting than that, so it is time to look at the implementation of attributes.

14.2.1. Default Attributes

When created, every kobject is given a set of default attributes. These attributes are specified by way of the kobj_type structure. That structure, remember, looks like this:

```
struct kobj_type {

    void (*release)(struct kobject *);

    struct sysfs_ops *sysfs_ops;

    struct attribute **default_attrs;

};
```

The default_attrs field lists the attributes to be created for every kobject of this type, and sysfs_ops provides the methods to implement those attributes. We start with default_attrs, which points to an array of pointers to attribute structures:

```
struct attribute {

    char *name;

    struct module *owner;

    mode_t mode;

};
```

14.3. Hotplug Event Generation

A *hotplug event* is a notification to user space from the kernel that something has changed in the system's configuration. They are generated whenever a kobject is created or destroyed. Such events are generated, for example, when a digital camera is plugged in with a USB cable, when a user switches console modes, or when a disk is repartitioned. Hotplug events turn into an invocation of */sbin/hotplug*, which can respond to each event by loading drivers, creating device nodes, mounting partitions, or taking any other action that is appropriate.

The last major kobject function we look at is the generation of these events. The actual event generation takes place when a kobject is passed to kobject_add or kobject_del. Before the event is handed to user space, code associated with the kobject (or, more specifically, the kset to which it belongs) has the opportunity to add information for user space or to disable event generation entirely.

14.3.1. Hotplug Operations

Actual control of hotplug events is exercised by way of a set of methods stored in the kset_hotplug_ops structure:

```
struct kset_hotplug_ops {

    int (*filter)(struct kset *kset, struct kobject *kobj);

    char *(*name)(struct kset *kset, struct kobject *kobj);

    int (*hotplug)(struct kset *kset, struct kobject *kobj,

                   char **envp, int num_envp, char *buffer,

                   int buffer_size);

};
```

A pointer to this structure is found in the hotplug_ops field of the kset structure. If a given kobject is not contained within a kset, the kernel searchs up through the hierarchy (via the parent pointer) until it finds a kobject that does have a kset; that kset's hotplug operations are then used.

The filter hotplug operation is called whenever the kernel is considering generating an event for a given kobject. If filter returns 0, the event is not created. This method, therefore, gives the kset code an opportunity to decide which events should be passed on to user space and which should not.

As an example of how this method might be used, consider the block subsystem. There are at least three types of kobjects used there, representing disks, partitions, and request queues. User space may want to react to the addition of a disk or a partition, but it does not normally care about request queues. So the filter method allows event generation only for kobjects representing disks and partitions. It looks like this:

```
static int block_hotplug_filter(struct kset *kset, struct kobject *kobj)

{

    struct kobj_type *ktype = get_ktype(kobj);



    return ((ktype =  = &ktype_block) || (ktype =  = &ktype_part));

}
```

Here, a quick test on the type of kobject is sufficient to decide whether the event should be generated or not.

When the user-space hotplug program is invoked, it is passed to the name of the relevant subsystem as its one and only parameter. The name hotplug method is charged with providing that name. It should return a simple string suitable for passing to user space.

Everything else that the hotplug script might want to know is passed in the environment. The final hotplug

14.4. Buses, Devices, and Drivers

So far, we have seen a great deal of low-level infrastructures and a relative shortage of examples. We try to make up for that in the rest of this chapter as we get into the higher levels of the Linux device model. To that end, we introduce a new virtual bus, which we call lddbus,[1] and modify the scullp driver to "connect" to that bus.

[1] The logical name for this bus, of course, would have been "sbus," but that name was already taken by a real, physical bus.

Once again, much of the material covered here will never be needed by many driver authors. Details at this level are generally handled at the bus level, and few authors need to add a new bus type. This information is useful, however, for anybody wondering what is happening inside the PCI, USB, etc. layers or who needs to make changes at that level.

14.4.1. Buses

A bus is a channel between the processor and one or more devices. For the purposes of the device model, all devices are connected via a bus, even if it is an internal, virtual, "platform" bus. Buses can plug into each other—a USB controller is usually a PCI device, for example. The device model represents the actual connections between buses and the devices they control.

In the Linux device model, a bus is represented by the bus_type structure, defined in <*linux/device.h*>. This structure looks like:

```
struct bus_type {

    char *name;

    struct subsystem subsys;

    struct kset drivers;

    struct kset devices;

    int (*match)(struct device *dev, struct device_driver *drv);

    struct device *(*add)(struct device * parent, char * bus_id);

    int (*hotplug) (struct device *dev, char **envp,

                    int num_envp, char *buffer, int buffer_size);

    /* Some fields omitted */

};
```

The name field is the name of the bus, something such as pci. You can see from the structure that each bus is its own subsystem; these subsystems do not live at the top level in sysfs, however. Instead, they are found underneath the bus subsystem. A bus contains two ksets, representing the known drivers for that bus and all devices plugged into the bus. Then, there is a set of methods that we will get to shortly.

14.4.1.1 Bus registration

As we mentioned, the example source includes a virtual bus implementation called lddbus. This bus sets up its bus_type structure as follows:

```
struct bus_type ldd_bus_type = {

    .name = "ldd",

    .match = ldd_match,

    .hotplug  = ldd_hotplug,

};
```

14.5. Classes

The final device model concept we examine in this chapter is the *class*. A class is a higher-level view of a device that abstracts out low-level implementation details. Drivers may see a SCSI disk or an ATA disk, but, at the class level, they are all simply disks. Classes allow user space to work with devices based on what they do, rather than how they are connected or how they work.

Almost all classes show up in sysfs under */sys/class*. Thus, for example, all network interfaces can be found under */sys/class/net*, regardless of the type of interface. Input devices can be found in */sys/class/input*, and serial devices are in */sys/class/tty*. The one exception is block devices, which can be found under */sys/block* for historical reasons.

Class membership is usually handled by high-level code without the need for explicit support from drivers. When the sbull driver (see Chapter 16) creates a virtual disk device, it automatically appears in */sys/block*. The snull network driver (see Chapter 17) does not have to do anything special for its interfaces to be represented in */sys/class/net*. There will be times, however, when drivers end up dealing with classes directly.

In many cases, the class subsystem is the best way of exporting information to user space. When a subsystem creates a class, it owns the class entirely, so there is no need to worry about which module owns the attributes found there. It also takes very little time wandering around in the more hardware-oriented parts of sysfs to realize that it can be an unfriendly place for direct browsing. Users more happily find information in */sys/class/some-widget* than under, say, */sys/devices/pci0000:00/0000:00:10.0/usb2/2-0:1.0*.

The driver core exports two distinct interfaces for managing classes. The class_simple routines are designed to make it as easy as possible to add new classes to the system; their main purpose, usually, is to expose attributes containing device numbers to enable the automatic creation of device nodes. The regular class interface is more complex but offers more features as well. We start with the simple version.

14.5.1. The class_simple Interface

The class_simple interface was intended to be so easy to use that nobody would have any excuse for not exporting, at a minimum, an attribute containing a device's assigned number. Using this interface is simply a matter of a couple of function calls, with little of the usual boilerplate associated with the Linux device model.

The first step is to create the class itself. That is accomplished with a call to class_simple_create:

```
struct class_simple *class_simple_create(struct module *owner, char *name);
```

This function creates a class with the given name. The operation can fail, of course, so the return value should always be checked (using IS_ERR, described in the Section 1.8 in Chapter 11) before continuing.

A simple class can be destroyed with:

```
void class_simple_destroy(struct class_simple *cs);
```

The real purpose of creating a simple class is to add devices to it; that task is achieved with:

```
struct class_device *class_simple_device_add(struct class_simple *cs,
                                             dev_t devnum,
                                             struct device *device,
                                             const char *fmt, ...);
```

Here, cs is the previously created simple class, devnum is the assigned device number, device is the struct device representing this device, and the remaining parameters are a printk-style format string and arguments to create the device name. This call adds an entry to the class containing one attribute, dev, which holds the device number. If the device parameter is not NULL, a symbolic link (called device) points to the device's entry under */sys/devices*.

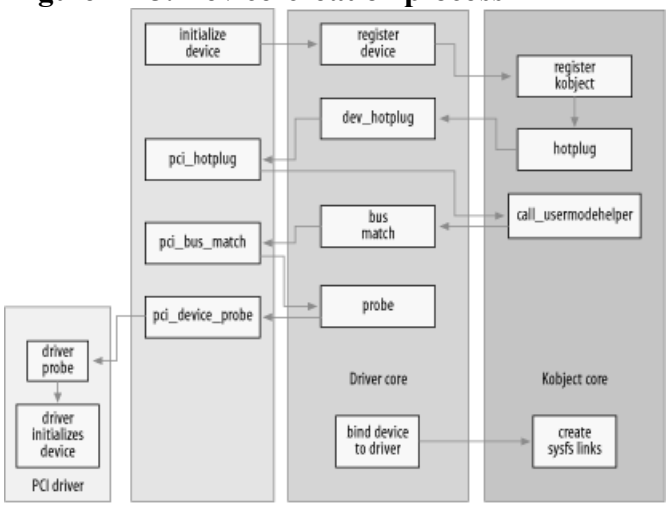
It is possible to add other attributes to a device entry. It is just a matter of using class_device_create_file, which we discuss in the next section with the rest of the full class subsystem.

14.6. Putting It All Together

To better understand what the driver model does, let us walk through the steps of a device's lifecycle within the kernel. We describe how the PCI subsystem interacts with the driver model, the basic concepts of how a driver is added and removed, and how a device is added and removed from the system. These details, while describing the PCI kernel code specifically, apply to all other subsystems that use the driver core to manage their drivers and devices.

The interaction between the PCI core, driver core, and the individual PCI drivers is quite complex, as Figure 14-2 shows.

Figure 14-3. Device-creation process



14.6.1. Add a Device

The PCI subsystem declares a single struct bus_type called pci_bus_type, which is initialized with the following values:

```
struct bus_type pci_bus_type = {

    .name       = "pci",

    .match      = pci_bus_match,

    .hotplug    = pci_hotplug,

    .suspend    = pci_device_suspend,

    .resume     = pci_device_resume,

    .dev_attrs = pci_dev_attrs,

};
```

This pci_bus_type variable is registered with the driver core when the PCI subsystem is loaded in the kernel with a call to bus_register. When that happens, the driver core creates a sysfs directory in */sys/bus/pci* that consists of two directories: *devices* and *drivers*.

All PCI drivers must define a struct pci_driver variable that defines the different functions that this PCI driver can do (for more information about the PCI subsystem and how to write a PCI driver, see Chapter 12). That structure contains a struct device_driver that is then initialized by the PCI core when the PCI driver is registered:

```
 /* initialize common driver fields */

drv->driver.name = drv->name;

drv->driver.bus = &pci_bus_type;
```

14.7. Hotplug

There are two different ways to view hotplugging. The kernel views hotplugging as an interaction between the hardware, the kernel, and the kernel driver. Users view hotplugging as the interaction between the kernel and user space through the program called /sbin/hotplug. This program is called by the kernel when it wants to notify user space that some type of hotplug event has just happened within the kernel.

14.7.1. Dynamic Devices

The most commonly used meaning of the term "hotplug" happens when discussing the fact that most all computer systems can now handle devices appearing or disappearing while the system is powered on. This is very different from the computer systems of only a few years ago, where the programmers knew that they needed to scan for all devices only at boot time, and they never had to worry about their devices disappearing until the power was turned off to the whole machine. Now, with the advent of USB, CardBus, PCMCIA, IEEE1394, and PCI Hotplug controllers, the Linux kernel needs to be able to reliably run no matter what hardware is added or removed from the system. This places an added burden on the device driver author, as they must now always handle a device being suddenly ripped out from underneath them without any notice.

Each different bus type handles the loss of a device in a different way. For example, when a PCI, CardBus, or PCMCIA device is removed from the system, it is usually a while before the driver is notified of this action through its remove function. Before that happens, all reads from the PCI bus return all bits set. This means that drivers need to always check the value of the data they read from the PCI bus and properly be able to handle a 0xff value.

An example of this can be seen in the drivers/usb/host/ehci-hcd.c driver, which is a PCI driver for a USB 2.0 (high-speed) controller card. It has the following code in its main handshake loop to detect if the controller card has been removed from the system:

```
 result = readl(ptr);

if (result =  = ~(u32)0)     /* card removed */

    return -ENODEV;
```

For USB drivers, when the device that a USB driver is bound to is removed from the system, any pending urbs that were submitted to the device start failing with the error -ENODEV. The driver needs to recognize this error and properly clean up any pending I/O if it occurs.

Hotpluggable devices are not limited only to traditional devices such as mice, keyboards, and network cards. There are numerous systems that now support removal and addition of entire CPUs and memory sticks. Fortunately the Linux kernel properly handles the addition and removal of such core "system" devices so that individual device drivers do not need to pay attention to these things.

14.7.2. The /sbin/hotplug Utility

As alluded to earlier in this chapter, whenever a device is added or removed from the system, a "hotplug event" is generated. This means that the kernel calls the user-space program /sbin/hotplug. This program is typically a very small bash script that merely passes execution on to a list of other programs that are placed in the /etc/hotplug.d/ directory tree. For most Linux distributions, this script looks like the following:

```
 DIR="/etc/hotplug.d"

for I in "${DIR}/$1/"*.hotplug "${DIR}/"default/*.hotplug ; do

    if [ -f $I ]; then

        test -x $I && $I $1 ;

    fi

done

exit 1
```

14.8. Dealing with Firmware

As a driver author, you may find yourself confronted with a device that must have firmware downloaded into it before it functions properly. The competition in many parts of the hardware market is so intense that even the cost of a bit of EEPROM for the device's controlling firmware is more than the manufacturer is willing to spend. So the firmware is distributed on a CD with the hardware, and the operating system is charged with conveying the firmware to the device itself.

You may be tempted to solve the firmware problem with a declaration like this:

```
static char my_firmware[  ] = { 0x34, 0x78, 0xa4, ... };
```

That approach is almost certainly a mistake, however. Coding firmware into a driver bloats the driver code, makes upgrading the firmware hard, and is very likely to run into licensing problems. It is highly unlikely that the vendor has released the firmware image under the GPL, so mixing it with GPL-licensed code is usually a mistake. For this reason, drivers containing wired-in firmware are unlikely to be accepted into the mainline kernel or included by Linux distributors.

14.8.1. The Kernel Firmware Interface

The proper solution is to obtain the firmware from user space when you need it. Please resist the temptation to try to open a file containing firmware directly from kernel space, however; that is an error-prone operation, and it puts policy (in the form of a file name) into the kernel. Instead, the correct approach is to use the firmware interface, which was created just for this purpose:

```
#include <linux/firmware.h>

int request_firmware(const struct firmware **fw, char *name,

                     struct device *device);
```

A call to request_firmware requests that user space locate and provide a firmware image to the kernel; we look at the details of how it works in a moment. The name should identify the firmware that is desired; the normal usage is the name of the firmware file as provided by the vendor. Something like *my_firmware.bin* is typical. If the firmware is successfully loaded, the return value is 0 (otherwise the usual error code is returned), and the fw argument is pointed to one of these structures:

```
struct firmware {

        size_t size;

        u8 *data;

};
```

That structure contains the actual firmware, which can now be downloaded to the device. Be aware that this firmware is unchecked data from user space; you should apply any and all tests you can think of to convince yourself that it is a proper firmware image before sending it to the hardware. Device firmware usually contains identification strings, checksums, and so on; check them all before trusting the data.

After you have sent the firmware to the device, you should release the in-kernel structure with:

```
void release_firmware(struct firmware *fw);
```

Since request_firmware asks user space to help, it is guaranteed to sleep before returning. If your driver is not in a position to sleep when it must ask for firmware, the asynchronous alternative may be used:

```
int request_firmware_nowait(struct module *module,

                            char *name, struct device *device, void *context,

                            void (*cont)(const struct firmware *fw, void *context));
```

The additional arguments here are module (which will almost always be THIS_MODULE), context (a private data pointer that is not used by the firmware subsystem), and cont. If all goes well, request_firmware_nowait begins the firmware load process and returns 0. At some future time, cont will be called with the result of the load. If the firmware load fails for some reason, fw is NULL.

14.9. Quick Reference

Many functions have been introduced in this chapter; here is a quick summary of them all.

14.9.1. Kobjects

```
#include <linux/kobject.h>
```

The include file containing definitions for kobjects, related structures, and functions.

```
void kobject_init(struct kobject *kobj);

int kobject_set_name(struct kobject *kobj, const char *format, ...);
```

Functions for kobject initialization.

```
struct kobject *kobject_get(struct kobject *kobj);

void kobject_put(struct kobject *kobj);
```

Functions that manage reference counts for kobjects.

```
struct kobj_type;

struct kobj_type *get_ktype(struct kobject *kobj);
```

Represents the type of structure within which a kobject is embedded. Use get_ktype to get the kobj_type associated with a given kobject.

```
int kobject_add(struct kobject *kobj);

extern int kobject_register(struct kobject *kobj);

void kobject_del(struct kobject *kobj);

void kobject_unregister(struct kobject *kobj);
```

kobject_add adds a kobject to the system, handling kset membership, sysfs representation, and hotplug event generation. kobject_register is a convenience function that combines kobject_init and kobject_add. Use kobject_del to remove a kobject or kobject_unregister, which combines kobject_del and kobject_put.

```
void kset_init(struct kset *kset);

int kset_add(struct kset *kset);

int kset_register(struct kset *kset);
```

Chapter 15. Memory Mapping and DMA

This chapter delves into the area of Linux memory management, with an emphasis on techniques that are useful to the device driver writer. Many types of driver programming require some understanding of how the virtual memory subsystem works; the material we cover in this chapter comes in handy more than once as we get into some of the more complex and performance-critical subsystems. The virtual memory subsystem is also a highly interesting part of the core Linux kernel and, therefore, it merits a look.

The material in this chapter is divided into three sections:

- The first covers the implementation of the mmap system call, which allows the mapping of device memory directly into a user process's address space. Not all devices require mmap support, but, for some, mapping device memory can yield significant performance improvements.
- We then look at crossing the boundary from the other direction with a discussion of direct access to user-space pages. Relatively few drivers need this capability; in many cases, the kernel performs this sort of mapping without the driver even being aware of it. But an awareness of how to map user-space memory into the kernel (with get_user_pages) can be useful.
- The final section covers direct memory access (DMA) I/O operations, which provide peripherals with direct access to system memory.

Of course, all of these techniques require an understanding of how Linux memory management works, so we start with an overview of that subsystem.

15.1. Memory Management in Linux

Rather than describing the theory of memory management in operating systems, this section tries to pinpoint the main features of the Linux implementation. Although you do not need to be a Linux virtual memory guru to implement mmap, a basic overview of how things work is useful. What follows is a fairly lengthy description of the data structures used by the kernel to manage memory. Once the necessary background has been covered, we can get into working with these structures.

15.1.1. Address Types

Linux is, of course, a virtual memory system, meaning that the addresses seen by user programs do not directly correspond to the physical addresses used by the hardware. Virtual memory introduces a layer of indirection that allows a number of nice things. With virtual memory, programs running on the system can allocate far more memory than is physically available; indeed, even a single process can have a virtual address space larger than the system's physical memory. Virtual memory also allows the program to play a number of tricks with the process's address space, including mapping the program's memory to device memory.

Thus far, we have talked about virtual and physical addresses, but a number of the details have been glossed over. The Linux system deals with several types of addresses, each with its own semantics. Unfortunately, the kernel code is not always very clear on exactly which type of address is being used in each situation, so the programmer must be careful.

The following is a list of address types used in Linux. Figure 15-1 shows how these address types relate to physical memory.

User virtual addresses

These are the regular addresses seen by user-space programs. User addresses are either 32 or 64 bits in length, depending on the underlying hardware architecture, and each process has its own virtual address space.

Physical addresses

The addresses used between the processor and the system's memory. Physical addresses are 32- or 64-bit quantities; even 32-bit systems can use larger physical addresses in some situations.

Bus addresses

The addresses used between peripheral buses and memory. Often, they are the same as the physical addresses used by the processor, but that is not necessarily the case. Some architectures can provide an I/O memory management unit (IOMMU) that remaps addresses between a bus and main memory. An IOMMU can make life easier in a number of ways (making a buffer scattered in memory appear contiguous to the device, for example), but programming the IOMMU is an extra step that must be performed when setting up DMA operations. Bus addresses are highly architecture dependent, of course.

Kernel logical addresses

These make up the normal address space of the kernel. These addresses map some portion (perhaps all) of main memory and are often treated as if they were physical addresses. On most architectures, logical addresses and their associated physical addresses differ only by a constant offset. Logical addresses use the hardware's native pointer size and, therefore, may be unable to address all of physical memory on heavily equipped 32-bit systems. Logical addresses are usually stored in variables of type unsigned long or void *. Memory returned from kmalloc has a kernel logical address.

Kernel virtual addresses

15.2. The mmap Device Operation

Memory mapping is one of the most interesting features of modern Unix systems. As far as drivers are concerned, memory mapping can be implemented to provide user programs with direct access to device memory.

A definitive example of mmap usage can be seen by looking at a subset of the virtual memory areas for the X Window System server:

```
cat /proc/731/maps

000a0000-000c0000 rwxs 000a0000 03:01 282652      /dev/mem

000f0000-00100000 r-xs 000f0000 03:01 282652      /dev/mem

00400000-005c0000 r-xp 00000000 03:01 1366927     /usr/X11R6/bin/Xorg

006bf000-006f7000 rw-p 001bf000 03:01 1366927     /usr/X11R6/bin/Xorg

2a95828000-2a958a8000 rw-s fcc00000 03:01 282652  /dev/mem

2a958a8000-2a9d8a8000 rw-s e8000000 03:01 282652  /dev/mem

...
```

The full list of the X server's VMAs is lengthy, but most of the entries are not of interest here. We do see, however, four separate mappings of */dev/mem*, which give some insight into how the X server works with the video card. The first mapping is at a0000, which is the standard location for video RAM in the 640-KB ISA hole. Further down, we see a large mapping at e8000000, an address which is above the highest RAM address on the system. This is a direct mapping of the video memory on the adapter.

These regions can also be seen in */proc/iomem*:

```
000a0000-000bffff : Video RAM area

000c0000-000ccfff : Video ROM

000d1000-000d1fff : Adapter ROM

000f0000-000fffff : System ROM

d7f00000-f7efffff : PCI Bus #01

  e8000000-effffffff : 0000:01:00.0

fc700000-fccfffff : PCI Bus #01

  fcc00000-fcc0ffff : 0000:01:00.0
```

Mapping a device means associating a range of user-space addresses to device memory. Whenever the program reads or writes in the assigned address range, it is actually accessing the device. In the X server example, using mmap allows quick and easy access to the video card's memory. For a performance-critical application like this, direct access makes a large difference.

As you might suspect, not every device lends itself to the mmap abstraction; it makes no sense, for instance, for serial ports and other stream-oriented devices. Another limitation of mmap is that mapping is PAGE_SIZE grained. The kernel can manage virtual addresses only at the level of page tables; therefore, the mapped area must be a multiple of PAGE_SIZE and must live in physical memory starting at an address that is a multiple of PAGE_SIZE. The kernel forces size granularity by making a region slightly bigger if its size isn't a multiple of the page size.

These limits are not a big constraint for drivers, because the program accessing the device is device dependent anyway. Since the program must know about how the device works, the programmer is not unduly bothered by the need to see to details like page alignment. A bigger constraint exists when ISA devices are used on some non-x86 platforms, because their hardware view of ISA may not be contiguous. For example, some Alpha

15.3. Performing Direct I/O

Most I/O operations are buffered through the kernel. The use of a kernel-space buffer allows a degree of separation between user space and the actual device; this separation can make programming easier and can also yield performance benefits in many situations. There are cases, however, where it can be beneficial to perform I/O directly to or from a user-space buffer. If the amount of data being transferred is large, transferring data directly without an extra copy through kernel space can speed things up.

One example of direct I/O use in the 2.6 kernel is the SCSI tape driver. Streaming tapes can pass a lot of data through the system, and tape transfers are usually record-oriented, so there is little benefit to buffering data in the kernel. So, when the conditions are right (the user-space buffer is page-aligned, for example), the SCSI tape driver performs its I/O without copying the data.

That said, it is important to recognize that direct I/O does not always provide the performance boost that one might expect. The overhead of setting up direct I/O (which involves faulting in and pinning down the relevant user pages) can be significant, and the benefits of buffered I/O are lost. For example, the use of direct I/O requires that the write system call operate synchronously; otherwise the application does not know when it can reuse its I/O buffer. Stopping the application until each write completes can slow things down, which is why applications that use direct I/O often use asynchronous I/O operations as well.

The real moral of the story, in any case, is that implementing direct I/O in a char driver is usually unnecessary and can be hurtful. You should take that step only if you are sure that the overhead of buffered I/O is truly slowing things down. Note also that block and network drivers need not worry about implementing direct I/O at all; in both cases, higher-level code in the kernel sets up and makes use of direct I/O when it is indicated, and driver-level code need not even know that direct I/O is being performed.

The key to implementing direct I/O in the 2.6 kernel is a function called get_user_pages , which is declared in <*linux/mm.h*> with the following prototype:

```
int get_user_pages(struct task_struct *tsk,

                   struct mm_struct *mm,

                   unsigned long start,

                   int len,

                   int write,

                   int force,

                   struct page **pages,

                   struct vm_area_struct **vmas);
```

This function has several arguments:

tsk

A pointer to the task performing the I/O; its main purpose is to tell the kernel who should be charged for any page faults incurred while setting up the buffer. This argument is almost always passed as current.

mm

A pointer to the memory management structure describing the address space to be mapped. The mm_struct structure is the piece that ties together all of the parts (VMAs) of a process's virtual address space. For driver use, this argument should always be current->mm.

start

15.4. Direct Memory Access

Direct memory access, or DMA , is the advanced topic that completes our overview of memory issues. DMA is the hardware mechanism that allows peripheral components to transfer their I/O data directly to and from main memory without the need to involve the system processor. Use of this mechanism can greatly increase throughput to and from a device, because a great deal of computational overhead is eliminated.

15.4.1. Overview of a DMA Data Transfer

Before introducing the programming details, let's review how a DMA transfer takes place, considering only input transfers to simplify the discussion.

Data transfer can be triggered in two ways: either the software asks for data (via a function such as read) or the hardware asynchronously pushes data to the system.

In the first case, the steps involved can be summarized as follows:

1. When a process calls read, the driver method allocates a DMA buffer and instructs the hardware to transfer its data into that buffer. The process is put to sleep.
2. The hardware writes data to the DMA buffer and raises an interrupt when it's done.
3. The interrupt handler gets the input data, acknowledges the interrupt, and awakens the process, which is now able to read data.

The second case comes about when DMA is used asynchronously. This happens, for example, with data acquisition devices that go on pushing data even if nobody is reading them. In this case, the driver should maintain a buffer so that a subsequent read call will return all the accumulated data to user space. The steps involved in this kind of transfer are slightly different:

1. The hardware raises an interrupt to announce that new data has arrived.
2. The interrupt handler allocates a buffer and tells the hardware where to transfer its data.
3. The peripheral device writes the data to the buffer and raises another interrupt when it's done.
4. The handler dispatches the new data, wakes any relevant process, and takes care of housekeeping.

A variant of the asynchronous approach is often seen with network cards. These cards often expect to see a circular buffer (often called a *DMA ring buffer*) established in memory shared with the processor; each incoming packet is placed in the next available buffer in the ring, and an interrupt is signaled. The driver then passes the network packets to the rest of the kernel and places a new DMA buffer in the ring.

The processing steps in all of these cases emphasize that efficient DMA handling relies on interrupt reporting. While it is possible to implement DMA with a polling driver, it wouldn't make sense, because a polling driver would waste the performance benefits that DMA offers over the easier processor-driven I/O.[4]

[4] There are, of course, exceptions to everything; see Section 15.2.6 for a demonstration of how high-performance network drivers are best implemented using polling.

Another relevant item introduced here is the DMA buffer. DMA requires device drivers to allocate one or more special buffers suited to DMA. Note that many drivers allocate their buffers at initialization time and use them

15.5. Quick Reference

This chapter introduced the following symbols related to memory handling.

15.5.1. Introductory Material

#include <linux/mm.h>

#include <asm/page.h>

Most of the functions and structures related to memory management are prototyped and defined in these header files.

void *_ _va(unsigned long physaddr);

unsigned long _ _pa(void *kaddr);

Macros that convert between kernel logical addresses and physical addresses.

PAGE_SIZE

PAGE_SHIFT

Constants that give the size (in bytes) of a page on the underlying hardware and the number of bits that a page frame number must be shifted to turn it into a physical address.

struct page

Structure that represents a hardware page in the system memory map.

struct page *virt_to_page(void *kaddr);

void *page_address(struct page *page);

struct page *pfn_to_page(int pfn);

Macros that convert between kernel logical addresses and their associated memory map entries. page_address works only for low-memory pages or high-memory pages that have been explicitly mapped. pfn_to_page converts a page frame number to its associated struct page pointer.

unsigned long kmap(struct page *page);

void kunmap(struct page *page);

kmap returns a kernel virtual address that is mapped to the given page, creating the mapping if need be. kunmap deletes the mapping for the given page.

Chapter 16. Block Drivers

So far, our discussion has been limited to char drivers. There are other types of drivers in Linux systems, however, and the time has come for us to widen our focus somewhat. Accordingly, this chapter discusses block drivers.

A block driver provides access to devices that transfer randomly accessible data in fixed-size blocks—disk drives, primarily. The Linux kernel sees block devices as being fundamentally different from char devices; as a result, block drivers have a distinct interface and their own particular challenges.

Efficient block drivers are critical for performance—and not just for explicit reads and writes in user applications. Modern systems with virtual memory work by shifting (hopefully) unneeded data to secondary storage, which is usually a disk drive. Block drivers are the conduit between core memory and secondary storage; therefore, they can be seen as making up part of the virtual memory subsystem. While it is possible to write a block driver without knowing about struct page and other important memory concepts, anybody needing to write a high-performance driver has to draw upon the material covered in Chapter 15.

Much of the design of the block layer is centered on performance. Many char devices can run below their maximum speed, and the performance of the system as a whole is not affected. The system cannot run well, however, if its block I/O subsystem is not well-tuned. The Linux block driver interface allows you to get the most out of a block device but imposes, necessarily, a degree of complexity that you must deal with. Happily, the 2.6 block interface is much improved over what was found in older kernels.

The discussion in this chapter is, as one would expect, centered on an example driver that implements a block-oriented, memory-based device. It is, essentially, a ramdisk. The kernel already contains a far superior ramdisk implementation, but our driver (called sbull) lets us demonstrate the creation of a block driver while minimizing unrelated complexity.

Before getting into the details, let's define a couple of terms precisely. A *block* is a fixed-size chunk of data, the size being determined by the kernel. Blocks are often 4096 bytes, but that value can vary depending on the architecture and the exact filesystem being used. A *sector*, in contrast, is a small block whose size is usually determined by the underlying hardware. The kernel expects to be dealing with devices that implement 512-byte sectors. If your device uses a different size, the kernel adapts and avoids generating I/O requests that the hardware cannot handle. It is worth keeping in mind, however, that any time the kernel presents you with a sector number, it is working in a world of 512-byte sectors. If you are using a different hardware sector size, you have to scale the kernel's sector numbers accordingly. We see how that is done in the sbull driver.

16.1. Registration

Block drivers, like char drivers, must use a set of registration interfaces to make their devices available to the kernel. The concepts are similar, but the details of block device registration are all different. You have a whole new set of data structures and device operations to learn.

16.1.1. Block Driver Registration

The first step taken by most block drivers is to register themselves with the kernel. The function for this task is register_blkdev (which is declared in <*linux/fs.h*>):

```
int register_blkdev(unsigned int major, const char *name);
```

The arguments are the major number that your device will be using and the associated name (which the kernel will display in */proc/devices*). If major is passed as 0, the kernel allocates a new major number and returns it to the caller. As always, a negative return value from register_blkdev indicates that an error has occurred.

The corresponding function for canceling a block driver registration is:

```
int unregister_blkdev(unsigned int major, const char *name);
```

Here, the arguments must match those passed to register_blkdev, or the function returns -EINVAL and not unregister anything.

In the 2.6 kernel, the call to register_blkdev is entirely optional. The functions performed by register_blkdev have been decreasing over time; the only tasks performed by this call at this point are (1) allocating a dynamic major number if requested, and (2) creating an entry in */proc/devices*. In future kernels, register_blkdev may be removed altogether. Meanwhile, however, most drivers still call it; it's traditional.

16.1.2. Disk Registration

While register_blkdev can be used to obtain a major number, it does not make any disk drives available to the system. There is a separate registration interface that you must use to manage individual drives. Using this interface requires familiarity with a pair of new structures, so that is where we start.

16.1.2.1 Block device operations

Char devices make their operations available to the system by way of the file_operations structure. A similar structure is used with block devices; it is struct block_device_operations, which is declared in <*linux/fs.h*>. The following is a brief overview of the fields found in this structure; we revisit them in more detail when we get into the details of the sbull driver:

int (*open)(struct inode *inode, struct file *filp);

int (*release)(struct inode *inode, struct file *filp);

Functions that work just like their char driver equivalents; they are called whenever the device is opened and closed. A block driver might respond to an open call by spinning up the device, locking the door (for removable media), etc. If you lock media into the device, you should certainly unlock it in the release method.

int (*ioctl)(struct inode *inode, struct file *filp, unsigned int cmd,

unsigned long arg);

Method that implements the ioctl system call. The block layer first intercepts a large number of standard requests, however; so most block driver ioctl methods are fairly short.

16.2. The Block Device Operations

We had a brief introduction to the block_device_operations structure in the previous section. Now we take some time to look at these operations in a bit more detail before getting into request processing. To that end, it is time to mention one other feature of the sbull driver: it pretends to be a removable device. Whenever the last user closes the device, a 30-second timer is set; if the device is not opened during that time, the contents of the device are cleared, and the kernel will be told that the media has been changed. The 30-second delay gives the user time to, for example, mount an sbull device after creating a filesystem on it.

16.2.1. The open and release Methods

To implement the simulated media removal, sbull must know when the last user has closed the device. A count of users is maintained by the driver. It is the job of the open and close methods to keep that count current.

The open method looks very similar to its char-driver equivalent; it takes the relevant inode and file structure pointers as arguments. When an inode refers to a block device, the field i_bdev->bd_disk contains a pointer to the associated gendisk structure; this pointer can be used to get to a driver's internal data structures for the device. That is, in fact, the first thing that the sbull open method does:

```
 static int sbull_open(struct inode *inode, struct file *filp)

{

    struct sbull_dev *dev = inode->i_bdev->bd_disk->private_data;



    del_timer_sync(&dev->timer);

    filp->private_data = dev;

    spin_lock(&dev->lock);

    if (! dev->users)

        check_disk_change(inode->i_bdev);

    dev->users++;

    spin_unlock(&dev->lock);

    return 0;

}
```

Once sbull_open has its device structure pointer, it calls del_timer_sync to remove the "media removal" timer, if any is active. Note that we do not lock the device spinlock until after the timer has been deleted; doing otherwise invites deadlock if the timer function runs before we can delete it. With the device locked, we call a kernel function called check_disk_change to check whether a media change has happened. One might argue that the kernel should make that call, but the standard pattern is for drivers to handle it at open time.

The last step is to increment the user count and return.

The task of the release method is, in contrast, to decrement the user count and, if indicated, start the media removal timer:

```
 static int sbull_release(struct inode *inode, struct file *filp)

{

    struct sbull_dev *dev = inode->i_bdev->bd_disk->private_data;



    spin_lock(&dev->lock);
```

16.3. Request Processing

The core of every block driver is its request function. This function is where the real work gets done—or at least started; all the rest is overhead. Consequently, we spend a fair amount of time looking at request processing in block drivers.

A disk driver's performance can be a critical part of the performance of the system as a whole. Therefore, the kernel's block subsystem has been written with performance very much in mind; it does everything possible to enable your driver to get the most out of the devices it controls. This is a good thing, in that it enables blindingly fast I/O. On the other hand, the block subsystem unnecessarily exposes a great deal of complexity in the driver API. It is possible to write a very simple request function (we will see one shortly), but if your driver must perform at a high level on complex hardware, it will be anything but simple.

16.3.1. Introduction to the request Method

The block driver request method has the following prototype:

```
void request(request_queue_t *queue);
```

This function is called whenever the kernel believes it is time for your driver to process some reads, writes, or other operations on the device. The request function does not need to actually complete all of the requests on the queue before it returns; indeed, it probably does not complete any of them for most real devices. It must, however, make a start on those requests and ensure that they are all, eventually, processed by the driver.

Every device has a request queue. This is because actual transfers to and from a disk can take place far away from the time the kernel requests them, and because the kernel needs the flexibility to schedule each transfer at the most propitious moment (grouping together, for instance, requests that affect sectors close together on the disk). And the request function, you may remember, is associated with a request queue when that queue is created. Let us look back at how sbull makes its queue:

```
dev->queue = blk_init_queue(sbull_request, &dev->lock);
```

Thus, when the queue is created, the request function is associated with it. We also provided a spinlock as part of the queue creation process. Whenever our request function is called, that lock is held by the kernel. As a result, the request function is running in an atomic context; it must follow all of the usual rules for atomic code discussed in Chapter 5.

The queue lock also prevents the kernel from queuing any other requests for your device while your request function holds the lock. Under some conditions, you may want to consider dropping that lock while the request function runs. If you do so, however, you must be sure not to access the request queue, or any other data structure protected by the lock, while the lock is not held. You must also reacquire the lock before the request function returns.

Finally, the invocation of the request function is (usually) entirely asynchronous with respect to the actions of any user-space process. You cannot assume that the kernel is running in the context of the process that initiated the current request. You do not know if the I/O buffer provided by the request is in kernel or user space. So any sort of operation that explicitly accesses user space is in error and will certainly lead to trouble. As you will see, everything your driver needs to know about the request is contained within the structures passed to you via the request queue.

16.3.2. A Simple request Method

The sbull example driver provides a few different methods for request processing. By default, sbull uses a method called sbull_request, which is meant to be an example of the simplest possible request method. Without further ado, here it is:

```
static void sbull_request(request_queue_t *q)

{

    struct request *req;
```

16.4. Some Other Details

This section covers a few other aspects of the block layer that may be of interest for advanced drivers. None of the following facilities need to be used to write a correct driver, but they may be helpful in some situations.

16.4.1. Command Pre-Preparation

The block layer provides a mechanism for drivers to examine and preprocess requests before they are returned from elv_next_request. This mechanism allows drivers to set up the actual drive commands ahead of time, decide whether the request can be handled at all, or perform other sorts of housekeeping.

If you want to use this feature, create a command preparation function that fits this prototype:

```
typedef int (prep_rq_fn) (request_queue_t *queue, struct request *req);
```

The request structure includes a field called cmd, which is an array of BLK_MAX_CDB bytes; this array may be used by the preparation function to store the actual hardware command (or any other useful information). This function should return one of the following values:

BLKPREP_OK

Command preparation went normally, and the request can be handed to your driver's request function.

BLKPREP_KILL

This request cannot be completed; it is failed with an error code.

BLKPREP_DEFER

This request cannot be completed at this time. It stays at the front of the queue but is not handed to the request function.

The preparation function is called by elv_next_request immediately before the request is returned to your driver. If this function returns BLKPREP_DEFER, the return value from elv_next_request to your driver is NULL. This mode of operation can be useful if, for example, your device has reached the maximum number of requests it can have outstanding.

To have the block layer call your preparation function, pass it to:

```
void blk_queue_prep_rq(request_queue_t *queue, prep_rq_fn *func);
```

By default, request queues have no preparation function.

16.4.2. Tagged Command Queueing

Hardware that can have multiple requests active at once usually supports some form of *tagged command queueing* (TCQ). TCQ is simply the technique of attaching an integer "tag" to each request so that when the drive completes one of those requests, it can tell the driver which one. In previous versions of the kernel, block drivers that implemented TCQ had to do all of the work themselves; in 2.6, a TCQ support infrastructure has been added to the block layer for all drivers to use.

If your drive performs tagged command queueing, you should inform the kernel of that fact at initialization time with a call to:

```
int blk_queue_init_tags(request_queue_t *queue, int depth,
                        struct blk_queue_tag *tags);
```

Here, queue is your request queue, and depth is the number of tagged requests your device can have outstanding at any given time. tags is an optional pointer to an array of struct blk_queue_tag structures; there must be depth of

16.5. Quick Reference

#include <linux/fs.h>

int register_blkdev(unsigned int major, const char *name);

int unregister_blkdev(unsigned int major, const char *name);

register_blkdev registers a block driver with the kernel and, optionally, obtains a major number. A driver can be unregistered with unregister_blkdev.

struct block_device_operations

Structure that holds most of the methods for block drivers.

#include <linux/genhd.h>

struct gendisk;

Structure that describes a single block device within the kernel.

struct gendisk *alloc_disk(int minors);

void add_disk(struct gendisk *gd);

Functions that allocate gendisk structures and return them to the system.

void set_capacity(struct gendisk *gd, sector_t sectors);

Stores the capacity of the device (in 512-byte sectors) within the gendisk structure.

void add_disk(struct gendisk *gd);

Adds a disk to the kernel. As soon as this function is called, your disk's methods can be invoked by the kernel.

int check_disk_change(struct block_device *bdev);

A kernel function that checks for a media change in the given disk drive and takes the required cleanup action when such a change is detected.

#include <linux/blkdev.h>

request_queue_t blk_init_queue(request_fn_proc *request, spinlock_t *lock);

void blk_cleanup_queue(request_queue_t *);

Chapter 17. Network Drivers

Having discussed char and block drivers, we are now ready to move on to the world of networking. Network interfaces are the third standard class of Linux devices, and this chapter describes how they interact with the rest of the kernel.

The role of a network interface within the system is similar to that of a mounted block device. A block device registers its disks and methods with the kernel, and then "transmits" and "receives" blocks on request, by means of its request function. Similarly, a network interface must register itself within specific kernel data structures in order to be invoked when packets are exchanged with the outside world.

There are a few important differences between mounted disks and packet-delivery interfaces. To begin with, a disk exists as a special file in the */dev* directory, whereas a network interface has no such entry point. The normal file operations (read, write, and so on) do not make sense when applied to network interfaces, so it is not possible to apply the Unix "everything is a file" approach to them. Thus, network interfaces exist in their own namespace and export a different set of operations.

Although you may object that applications use the read and write system calls when using sockets, those calls act on a software object that is distinct from the interface. Several hundred sockets can be multiplexed on the same physical interface.

But the most important difference between the two is that block drivers operate only in response to requests from the kernel, whereas network drivers receive packets asynchronously from the outside. Thus, while a block driver is asked to send a buffer toward the kernel, the network device asks to push incoming packets toward the kernel. The kernel interface for network drivers is designed for this different mode of operation.

Network drivers also have to be prepared to support a number of administrative tasks, such as setting addresses, modifying transmission parameters, and maintaining traffic and error statistics. The API for network drivers reflects this need and, therefore, looks somewhat different from the interfaces we have seen so far.

The network subsystem of the Linux kernel is designed to be completely protocol-independent. This applies to both networking protocols (Internet protocol [IP] versus IPX or other protocols) and hardware protocols (Ethernet versus token ring, etc.). Interaction between a network driver and the kernel properly deals with one network packet at a time; this allows protocol issues to be hidden neatly from the driver and the physical transmission to be hidden from the protocol.

This chapter describes how the network interfaces fit in with the rest of the Linux kernel and provides examples in the form of a memory-based modularized network interface, which is called (you guessed it) snull. To simplify the discussion, the interface uses the Ethernet hardware protocol and transmits IP packets. The knowledge you acquire from examining snull can be readily applied to protocols other than IP, and writing a non-Ethernet driver is different only in tiny details related to the actual network protocol.

This chapter doesn't talk about IP numbering schemes, network protocols, or other general networking concepts. Such topics are not (usually) of concern to the driver writer, and it's impossible to offer a satisfactory overview of networking technology in less than a few hundred pages. The interested reader is urged to refer to other books describing networking issues.

One note on terminology is called for before getting into network devices. The networking world uses the term octet to refer to a group of eight bits, which is generally the smallest unit understood by networking devices and protocols. The term byte is almost never encountered in this context. In keeping with standard usage, we will use octet when talking about networking devices.

The term "header" also merits a quick mention. A header is a set of bytes (err, octets) prepended to a packet as it is passed through the various layers of the networking subsystem. When an application sends a block of data through a TCP socket, the networking subsystem breaks that data up into packets and puts a TCP header, describing where each packet fits within the stream, at the beginning. The lower levels then put an IP header, used to route the packet to its destination, in front of the TCP header. If the packet moves over an Ethernet-like medium, an Ethernet header, interpreted by the hardware, goes in front of the rest. Network drivers need not concern themselves with higher-level headers (usually), but they often must be involved in the creation of the hardware-level header.

17.1. How snull Is Designed

This section discusses the design concepts that led to the snull network interface. Although this information might appear to be of marginal use, failing to understand it might lead to problems when you play with the sample code.

The first, and most important, design decision was that the sample interfaces should remain independent of real hardware, just like most of the sample code used in this book. This constraint led to something that resembles the loopback interface. snull is not a loopback interface; however, it simulates conversations with real remote hosts in order to better demonstrate the task of writing a network driver. The Linux loopback driver is actually quite simple; it can be found in *drivers/net/loopback.c*.

Another feature of snull is that it supports only IP traffic. This is a consequence of the internal workings of the interface—snull has to look inside and interpret the packets to properly emulate a pair of hardware interfaces. Real interfaces don't depend on the protocol being transmitted, and this limitation of snull doesn't affect the fragments of code shown in this chapter.

17.1.1. Assigning IP Numbers

The snull module creates two interfaces. These interfaces are different from a simple loopback, in that whatever you transmit through one of the interfaces loops back to the other one, not to itself. It looks like you have two external links, but actually your computer is replying to itself.

Unfortunately, this effect can't be accomplished through IP number assignments alone, because the kernel wouldn't send out a packet through interface A that was directed to its own interface B. Instead, it would use the loopback channel without passing through snull. To be able to establish a communication through the snull interfaces, the source and destination addresses need to be modified during data transmission. In other words, packets sent through one of the interfaces should be received by the other, but the receiver of the outgoing packet shouldn't be recognized as the local host. The same applies to the source address of received packets.

To achieve this kind of "hidden loopback," the snull interface toggles the least significant bit of the third octet of both the source and destination addresses; that is, it changes both the network number and the host number of class C IP numbers. The net effect is that packets sent to network A (connected to sn0, the first interface) appear on the sn1 interface as packets belonging to network B.

To avoid dealing with too many numbers, let's assign symbolic names to the IP numbers involved:

- snullnet0 is the network that is connected to the sn0 interface. Similarly, snullnet1 is the network connected to sn1. The addresses of these networks should differ only in the least significant bit of the third octet. These networks must have 24-bit netmasks.
- local0 is the IP address assigned to the sn0 interface; it belongs to snullnet0. The address associated with sn1 is local1. local0 and local1 must differ in the least significant bit of their third octet and in the fourth octet.
- remote0 is a host in snullnet0, and its fourth octet is the same as that of local1. Any packet sent to remote0 reaches local1 after its network address has been modified by the interface code. The host remote1 belongs to snullnet1, and its fourth octet is the same as that of local0.

The operation of the snull interfaces is depicted in Figure 17-1, in which the hostname associated with each interface is printed near the interface name.

Figure 17-1. How a host sees its interfaces

17.2. Connecting to the Kernel

We start looking at the structure of network drivers by dissecting the snull source. Keeping the source code for several drivers handy might help you follow the discussion and to see how real-world Linux network drivers operate. As a place to start, we suggest *loopback.c*, *plip.c*, and *e100.c*, in order of increasing complexity. All these files live in *drivers/net*, within the kernel source tree.

17.2.1. Device Registration

When a driver module is loaded into a running kernel, it requests resources and offers facilities; there's nothing new in that. And there's also nothing new in the way resources are requested. The driver should probe for its device and its hardware location (I/O ports and IRQ line)—but not register them—as described in Section 10.2. The way a network driver is registered by its module initialization function is different from char and block drivers. Since there is no equivalent of major and minor numbers for network interfaces, a network driver does not request such a number. Instead, the driver inserts a data structure for each newly detected interface into a global list of network devices.

Each interface is described by a struct net_device item, which is defined in <linux/netdevice.h>. The snull driver keeps pointers to two of these structures (for sn0 and sn1) in a simple array:

```
struct net_device *snull_devs[2];
```

The net_device structure, like many other kernel structures, contains a kobject and is, therefore, reference-counted and exported via sysfs. As with other such structures, it must be allocated dynamically. The kernel function provided to perform this allocation is alloc_netdev, which has the following prototype:

```
struct net_device *alloc_netdev(int sizeof_priv,
                                const char *name,
                                void (*setup)(struct net_device *));
```

Here, sizeof_priv is the size of the driver's "private data" area; with network devices, that area is allocated along with the net_device structure. In fact, the two are allocated together in one large chunk of memory, but driver authors should pretend that they don't know that. name is the name of this interface, as is seen by user space; this name can have a printf-style %d in it. The kernel replaces the %d with the next available interface number. Finally, setup is a pointer to an initialization function that is called to set up the rest of the net_device structure. We get to the initialization function shortly, but, for now, suffice it to say that snull allocates its two device structures in this way:

```
snull_devs[0] = alloc_netdev(sizeof(struct snull_priv), "sn%d",
        snull_init);
snull_devs[1] = alloc_netdev(sizeof(struct snull_priv), "sn%d",
        snull_init);
if (snull_devs[0] =  = NULL || snull_devs[1] =  = NULL)
    goto out;
```

As always, we must check the return value to ensure that the allocation succeeded.

The networking subsystem provides a number of helper functions wrapped around alloc_netdev for various types of interfaces. The most common is alloc_etherdev, which is defined in <linux/etherdevice.h>:

```
struct net_device *alloc_etherdev(int sizeof_priv);
```

This function allocates a network device using eth%d for the name argument. It provides its own initialization function (ether_setup) that sets several net_device fields with appropriate values for Ethernet devices. Thus, there is no driver-supplied initialization function for alloc_etherdev; the driver should simply do its required initialization directly after a successful allocation. Writers of drivers for other types of devices may want to take advantage of one of the other helper functions, such as alloc_fcdev (defined in <linux/fcdevice.h>) for fiber-channel devices, alloc_fddidev (<linux/fddidevice.h>) for FDDI devices, or alloc_trdev (<linux/trdevice.h>) for token ring devices.

17.3. The net_device Structure in Detail

The net_device structure is at the very core of the network driver layer and deserves a complete description. This list describes all the fields, but more to provide a reference than to be memorized. The rest of this chapter briefly describes each field as soon as it is used in the sample code, so you don't need to keep referring back to this section.

17.3.1. Global Information

The first part of struct net_device is composed of the following fields:

char name[IFNAMSIZ];

The name of the device. If the name set by the driver contains a %d format string, register_netdev replaces it with a number to make a unique name; assigned numbers start at 0.

unsigned long state;

Device state. The field includes several flags. Drivers do not normally manipulate these flags directly; instead, a set of utility functions has been provided. These functions are discussed shortly when we get into driver operations.

struct net_device *next;

Pointer to the next device in the global linked list. This field shouldn't be touched by the driver.

int (*init)(struct net_device *dev);

An initialization function. If this pointer is set, the function is called by register_netdev to complete the initialization of the net_device structure. Most modern network drivers do not use this function any longer; instead, initialization is performed before registering the interface.

17.3.2. Hardware Information

The following fields contain low-level hardware information for relatively simple devices. They are a holdover from the earlier days of Linux networking; most modern drivers do make use of them (with the possible exception of if_port). We list them here for completeness.

unsigned long rmem_end;

unsigned long rmem_start;

unsigned long mem_end;

unsigned long mem_start;

Device memory information. These fields hold the beginning and ending addresses of the shared memory used by the device. If the device has different receive and transmit memories, the mem fields are used for transmit memory and the rmem fields for receive memory. The rmem fields are never referenced outside of the driver itself. By convention, the end fields are set so that end - start is the amount of available onboard memory.

17.4. Opening and Closing

Our driver can probe for the interface at module load time or at kernel boot. Before the interface can carry packets, however, the kernel must open it and assign an address to it. The kernel opens or closes an interface in response to the ifconfig command.

When ifconfig is used to assign an address to the interface, it performs two tasks. First, it assigns the address by means of ioctl(SIOCSIFADDR) (Socket I/O Control Set Interface Address). Then it sets the IFF_UP bit in dev->flag by means of ioctl(SIOCSIFFLAGS) (Socket I/O Control Set Interface Flags) to turn the interface on.

As far as the device is concerned, ioctl(SIOCSIFADDR) does nothing. No driver function is invoked—the task is device independent, and the kernel performs it. The latter command (ioctl(SIOCSIFFLAGS)), however, calls the open method for the device.

Similarly, when the interface is shut down, ifconfig uses ioctl(SIOCSIFFLAGS) to clear IFF_UP, and the stop method is called.

Both device methods return 0 in case of success and the usual negative value in case of error.

As far as the actual code is concerned, the driver has to perform many of the same tasks as the char and block drivers do. open requests any system resources it needs and tells the interface to come up; stop shuts down the interface and releases system resources. Network drivers must perform some additional steps at open time, however.

First, the hardware (MAC) address needs to be copied from the hardware device to dev->dev_addr before the interface can communicate with the outside world. The hardware address can then be copied to the device at open time. The snull software interface assigns it from within open; it just fakes a hardware number using an ASCII string of length ETH_ALEN, the length of Ethernet hardware addresses.

The open method should also start the interface's transmit queue (allowing it to accept packets for transmission) once it is ready to start sending data. The kernel provides a function to start the queue:

```
void netif_start_queue(struct net_device *dev);
```

The open code for snull looks like the following:

```
int snull_open(struct net_device *dev)

{

    /* request_region(  ), request_irq(  ), ....  (like fops->open) */



    /*

     * Assign the hardware address of the board: use "\0SNULx", where

     * x is 0 or 1. The first byte is '\0' to avoid being a multicast

     * address (the first byte of multicast addrs is odd).

     */

    memcpy(dev->dev_addr, "\0SNUL0", ETH_ALEN);

    if (dev =  = snull_devs[1])

        dev->dev_addr[ETH_ALEN-1]++; /* \0SNUL1 */

    netif_start_queue(dev);

    return 0;

}
```

17.5. Packet Transmission

The most important tasks performed by network interfaces are data transmission and reception. We start with transmission because it is slightly easier to understand.

Transmission refers to the act of sending a packet over a network link. Whenever the kernel needs to transmit a data packet, it calls the driver's hard_start_transmit method to put the data on an outgoing queue. Each packet handled by the kernel is contained in a socket buffer structure (struct sk_buff), whose definition is found in <linux/skbuff.h>. The structure gets its name from the Unix abstraction used to represent a network connection, the socket. Even if the interface has nothing to do with sockets, each network packet belongs to a socket in the higher network layers, and the input/output buffers of any socket are lists of struct sk_buff structures. The same sk_buff structure is used to host network data throughout all the Linux network subsystems, but a socket buffer is just a packet as far as the interface is concerned.

A pointer to sk_buff is usually called skb, and we follow this practice both in the sample code and in the text.

The socket buffer is a complex structure, and the kernel offers a number of functions to act on it. The functions are described later in Section 17.10; for now, a few basic facts about sk_buff are enough for us to write a working driver.

The socket buffer passed to hard_start_xmit contains the physical packet as it should appear on the media, complete with the transmission-level headers. The interface doesn't need to modify the data being transmitted. skb->data points to the packet being transmitted, and skb->len is its length in octets. This situation gets a little more complicated if your driver can handle scatter/gather I/O; we get to that in Section 17.5.3.

The snull packet transmission code follows; the physical transmission machinery has been isolated in another function, because every interface driver must implement it according to the specific hardware being driven:

```
int snull_tx(struct sk_buff *skb, struct net_device *dev)

{

    int len;

    char *data, shortpkt[ETH_ZLEN];

    struct snull_priv *priv = netdev_priv(dev);



    data = skb->data;

    len = skb->len;

    if (len < ETH_ZLEN) {

        memset(shortpkt, 0, ETH_ZLEN);

        memcpy(shortpkt, skb->data, skb->len);

        len = ETH_ZLEN;

        data = shortpkt;

    }

    dev->trans_start = jiffies; /* save the timestamp */



    /* Remember the skb, so we can free it at interrupt time */

    priv->skb = skb;
```

17.6. Packet Reception

Receiving data from the network is trickier than transmitting it, because an sk_buff must be allocated and handed off to the upper layers from within an atomic context. There are two modes of packet reception that may be implemented by network drivers: interrupt driven and polled. Most drivers implement the interrupt-driven technique, and that is the one we cover first. Some drivers for high-bandwidth adapters may also implement the polled technique; we look at this approach in the Section 17.8.

The implementation of snull separates the "hardware" details from the device-independent housekeeping. Therefore, the function snull_rx is called from the snull "interrupt" handler after the hardware has received the packet, and it is already in the computer's memory. snull_rx receives a pointer to the data and the length of the packet; its sole responsibility is to send the packet and some additional information to the upper layers of networking code. This code is independent of the way the data pointer and length are obtained.

```
 void snull_rx(struct net_device *dev, struct snull_packet *pkt)

{

    struct sk_buff *skb;

    struct snull_priv *priv = netdev_priv(dev);



    /*

     * The packet has been retrieved from the transmission

     * medium. Build an skb around it, so upper layers can handle it

     */

    skb = dev_alloc_skb(pkt->datalen + 2);

    if (!skb) {

        if (printk_ratelimit(  ))

            printk(KERN_NOTICE "snull rx: low on mem - packet dropped\n");

        priv->stats.rx_dropped++;

        goto out;

    }

    memcpy(skb_put(skb, pkt->datalen), pkt->data, pkt->datalen);



    /* Write metadata, and then pass to the receive level */

    skb->dev = dev;

    skb->protocol = eth_type_trans(skb, dev);

    skb->ip_summed = CHECKSUM_UNNECESSARY; /* don't check it */

    priv->stats.rx_packets++;

    priv->stats.rx_bytes += pkt->datalen;

    netif_rx(skb);

  out:
```

17.7. The Interrupt Handler

Most hardware interfaces are controlled by means of an interrupt handler. The hardware interrupts the processor to signal one of two possible events: a new packet has arrived or transmission of an outgoing packet is complete. Network interfaces can also generate interrupts to signal errors, link status changes, and so on.

The usual interrupt routine can tell the difference between a new-packet-arrived interrupt and a done-transmitting notification by checking a status register found on the physical device. The snull interface works similarly, but its status word is implemented in software and lives in dev->priv. The interrupt handler for a network interface looks like this:

```
static void snull_regular_interrupt(int irq, void *dev_id, struct pt_regs *regs)

{

    int statusword;

    struct snull_priv *priv;

    struct snull_packet *pkt = NULL;

    /*

     * As usual, check the "device" pointer to be sure it is

     * really interrupting.

     * Then assign "struct device *dev"

     */

    struct net_device *dev = (struct net_device *)dev_id;

    /* ... and check with hw if it's really ours */



    /* paranoid */

    if (!dev)

        return;



    /* Lock the device */

    priv = netdev_priv(dev);

    spin_lock(&priv->lock);



    /* retrieve statusword: real netdevices use I/O instructions */

    statusword = priv->status;

    priv->status = 0;

    if (statusword & SNULL_RX_INTR) {

        /* send it to snull_rx for handling */

        pkt = priv->rx_queue;

        if (pkt) {
```

17.8. Receive Interrupt Mitigation

When a network driver is written as we have described above, the processor is interrupted for every packet received by your interface. In many cases, that is the desired mode of operation, and it is not a problem. High-bandwidth interfaces, however, can receive thousands of packets per second. With that sort of interrupt load, the overall performance of the system can suffer.

As a way of improving the performance of Linux on high-end systems, the networking subsystem developers have created an alternative interface (called NAPI)[1] based on polling. "Polling" can be a dirty word among driver developers, who often see polling techniques as inelegant and inefficient. Polling is inefficient, however, only if the interface is polled when there is no work to do. When the system has a high-speed interface handling heavy traffic, there is always more packets to process. There is no need to interrupt the processor in such situations; it is enough that the new packets be collected from the interface every so often.

[1] NAPI stands for "new API"; the networking hackers are better at creating interfaces than naming them.

Stopping receive interrupts can take a substantial amount of load off the processor. NAPI-compliant drivers can also be told not to feed packets into the kernel if those packets are just dropped in the networking code due to congestion, which can also help performance when that help is needed most. For various reasons, NAPI drivers are also less likely to reorder packets.

Not all devices can operate in the NAPI mode, however. A NAPI-capable interface must be able to store several packets (either on the card itself, or in an in-memory DMA ring). The interface should be capable of disabling interrupts for received packets, while continuing to interrupt for successful transmissions and other events. There are other subtle issues that can make writing a NAPI-compliant driver harder; see *Documentation/networking/NAPI_HOWTO.txt* in the kernel source tree for the details.

Relatively few drivers implement the NAPI interface. If you are writing a driver for an interface that may generate a huge number of interrupts, however, taking the time to implement NAPI may well prove worthwhile.

The snull driver, when loaded with the use_napi parameter set to a nonzero value, operates in the NAPI mode. At initialization time, we have to set up a couple of extra struct net_device fields:

```
 if (use_napi) {

    dev->poll        = snull_poll;

    dev->weight      = 2;

}
```

The poll field must be set to your driver's polling function; we look at snull_poll shortly. The weight field describes the relative importance of the interface: how much traffic should be accepted from the interface when resources are tight. There are no strict rules for how the weight parameter should be set; by convention, 10 MBps Ethernet interfaces set weight to 16, while faster interfaces use 64. You should not set weight to a value greater than the number of packets your interface can store. In snull, we set the weight to two as a way of demonstrating deferred packet reception.

The next step in the creation of a NAPI-compliant driver is to change the interrupt handler. When your interface (which should start with receive interrupts enabled) signals that a packet has arrived, the interrupt handler should not process that packet. Instead, it should disable further receive interrupts and tell the kernel that it is time to start polling the interface. In the snull "interrupt" handler, the code that responds to packet reception interrupts has been changed to the following:

```
 if (statusword & SNULL_RX_INTR) {

    snull_rx_ints(dev, 0);  /* Disable further interrupts */

    netif_rx_schedule(dev);

}
```

When the interface tells us that a packet is available, the interrupt handler leaves it in the interface; all that needs

17.9. Changes in Link State

Network connections, by definition, deal with the world outside the local system. Therefore, they are often affected by outside events, and they can be transient things. The networking subsystem needs to know when network links go up or down, and it provides a few functions that the driver may use to convey that information.

Most networking technologies involving an actual, physical connection provide a *carrier* state; the presence of the carrier means that the hardware is present and ready to function. Ethernet adapters, for example, sense the carrier signal on the wire; when a user trips over the cable, that carrier vanishes, and the link goes down. By default, network devices are assumed to have a carrier signal present. The driver can change that state explicitly, however, with these functions:

```
void netif_carrier_off(struct net_device *dev);

void netif_carrier_on(struct net_device *dev);
```

If your driver detects a lack of carrier on one of its devices, it should call netif_carrier_off to inform the kernel of this change. When the carrier returns, netif_carrier_on should be called. Some drivers also call netif_carrier_off when making major configuration changes (such as media type); once the adapter has finished resetting itself, the new carrier is detected and traffic can resume.

An integer function also exists:

```
int netif_carrier_ok(struct net_device *dev);
```

This can be used to test the current carrier state (as reflected in the device structure).

17.10. The Socket Buffers

We've now covered most of the issues related to network interfaces. What's still missing is some more detailed discussion of the sk_buff structure. The structure is at the core of the network subsystem of the Linux kernel, and we now introduce both the main fields of the structure and the functions used to act on it.

Although there is no strict need to understand the internals of sk_buff, the ability to look at its contents can be helpful when you are tracking down problems and when you are trying to optimize your code. For example, if you look in *loopback.c*, you'll find an optimization based on knowledge of the sk_buff internals. The usual warning applies here: if you write code that takes advantage of knowledge of the sk_buff structure, you should be prepared to see it break with future kernel releases. Still, sometimes the performance advantages justify the additional maintenance cost.

We are not going to describe the whole structure here, just the fields that might be used from within a driver. If you want to see more, you can look at <linux/skbuff.h>, where the structure is defined and the functions are prototyped. Additional details about how the fields and functions are used can be easily retrieved by grepping in the kernel sources.

17.10.1. The Important Fields

The fields introduced here are the ones a driver might need to access. They are listed in no particular order.

struct net_device *dev;

The device receiving or sending this buffer.

union { /* ... */ } h;

union { /* ... */ } nh;

union { /*... */} mac;

Pointers to the various levels of headers contained within the packet. Each field of the union is a pointer to a different type of data structure. h hosts pointers to transport layer headers (for example, struct tcphdr *th); nh includes network layer headers (such as struct iphdr *iph); and mac collects pointers to link-layer headers (such as struct ethdr *ethernet).

If your driver needs to look at the source and destination addresses of a TCP packet, it can find them in skb->h.th. See the header file for the full set of header types that can be accessed in this way.

Note that network drivers are responsible for setting the mac pointer for incoming packets. This task is normally handled by eth_type_trans, but non-Ethernet drivers have to set skb->mac.raw directly, as shown in Section 17.11.3.

unsigned char *head;

unsigned char *data;

unsigned char *tail;

unsigned char *end;

17.11. MAC Address Resolution

An interesting issue with Ethernet communication is how to associate the MAC addresses (the interface's unique hardware ID) with the IP number. Most protocols have a similar problem, but we concentrate on the Ethernet-like case here. We try to offer a complete description of the issue, so we show three situations: ARP, Ethernet headers without ARP (such as plip), and non-Ethernet headers.

17.11.1. Using ARP with Ethernet

The usual way to deal with address resolution is by using the Address Resolution Protocol (ARP). Fortunately, ARP is managed by the kernel, and an Ethernet interface doesn't need to do anything special to support ARP. As long as dev->addr and dev->addr_len are correctly assigned at open time, the driver doesn't need to worry about resolving IP numbers to MAC addresses; ether_setup assigns the correct device methods to dev->hard_header and dev->rebuild_header.

Although the kernel normally handles the details of address resolution (and caching of the results), it calls upon the interface driver to help in the building of the packet. After all, the driver knows about the details of the physical layer header, while the authors of the networking code have tried to insulate the rest of the kernel from that knowledge. To this end, the kernel calls the driver's hard_header method to lay out the packet with the results of the ARP query. Normally, Ethernet driver writers need not know about this process—the common Ethernet code takes care of everything.

17.11.2. Overriding ARP

Simple point-to-point network interfaces, such as plip, might benefit from using Ethernet headers, while avoiding the overhead of sending ARP packets back and forth. The sample code in snull also falls into this class of network devices. snull cannot use ARP because the driver changes IP addresses in packets being transmitted, and ARP packets exchange IP addresses as well. Although we could have implemented a simple ARP reply generator with little trouble, it is more illustrative to show how to handle physical-layer headers directly.

If your device wants to use the usual hardware header without running ARP, you need to override the default dev->hard_header method. This is how snull implements it, as a very short function:

```
int snull_header(struct sk_buff *skb, struct net_device *dev,

                unsigned short type, void *daddr, void *saddr,

                unsigned int len)

{

    struct ethhdr *eth = (struct ethhdr *)skb_push(skb,ETH_HLEN);



    eth->h_proto = htons(type);

    memcpy(eth->h_source, saddr ? saddr : dev->dev_addr, dev->addr_len);

    memcpy(eth->h_dest,   daddr ? daddr : dev->dev_addr, dev->addr_len);

    eth->h_dest[ETH_ALEN-1]   ^= 0x01;   /* dest is us xor 1 */

    return (dev->hard_header_len);

}
```

The function simply takes the information provided by the kernel and formats it into a standard Ethernet header. It also toggles a bit in the destination Ethernet address, for reasons described later.

When a packet is received by the interface, the hardware header is used in a couple of ways by eth_type_trans. We have already seen this call in snull_rx:

```
skb->protocol = eth_type_trans(skb, dev);
```

17.12. Custom ioctl Commands

We have seen that the ioctl system call is implemented for sockets; SIOCSIFADDR and SIOCSIFMAP are examples of "socket ioctls." Now let's see how the third argument of the system call is used by networking code.

When the ioctl system call is invoked on a socket, the command number is one of the symbols defined in <linux/sockios.h>, and the sock_ioctl function directly invokes a protocol-specific function (where "protocol" refers to the main network protocol being used, for example, IP or AppleTalk).

Any ioctl command that is not recognized by the protocol layer is passed to the device layer. These device-related ioctl commands accept a third argument from user space, a struct ifreq *. This structure is defined in <linux/if.h>. The SIOCSIFADDR and SIOCSIFMAP commands actually work on the ifreq structure. The extra argument to SIOCSIFMAP, although defined as ifmap, is just a field of ifreq.

In addition to using the standardized calls, each interface can define its own ioctl commands. The plip interface, for example, allows the interface to modify its internal timeout values via ioctl. The ioctl implementation for sockets recognizes 16 commands as private to the interface: SIOCDEVPRIVATE through SIOCDEVPRIVATE+15.[2]

[2] Note that, according to <linux/sockios.h>, the SIOCDEVPRIVATE commands are deprecated. What should replace them is not clear, however, and numerous in-tree drivers still use them.

When one of these commands is recognized, dev->do_ioctl is called in the relevant interface driver. The function receives the same struct ifreq * pointer that the general-purpose ioctl function uses:

```
int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
```

The ifr pointer points to a kernel-space address that holds a copy of the structure passed by the user. After do_ioctl returns, the structure is copied back to user space; Therefore, the driver can use the private commands to both receive and return data.

The device-specific commands can choose to use the fields in struct ifreq, but they already convey a standardized meaning, and it's unlikely that the driver can adapt the structure to its needs. The field ifr_data is a caddr_t item (a pointer) that is meant to be used for device-specific needs. The driver and the program used to invoke its ioctl commands should agree about the use of ifr_data. For example, pppstats uses device-specific commands to retrieve information from the ppp interface driver.

It's not worth showing an implementation of do_ioctl here, but with the information in this chapter and the kernel examples, you should be able to write one when you need it. Note, however, that the plip implementation uses ifr_data incorrectly and should not be used as an example for an ioctl implementation.

17.13. Statistical Information

The last method a driver needs is get_stats. This method returns a pointer to the statistics for the device. Its implementation is pretty easy; the one shown works even when several interfaces are managed by the same driver, because the statistics are hosted within the device data structure.

```
struct net_device_stats *snull_stats(struct net_device *dev)

{

    struct snull_priv *priv = netdev_priv(dev);

    return &priv->stats;

}
```

The real work needed to return meaningful statistics is distributed throughout the driver, where the various fields are updated. The following list shows the most interesting fields in struct net_device_stats:

unsigned long rx_packets;

unsigned long tx_packets;

The total number of incoming and outgoing packets successfully transferred by the interface.

unsigned long rx_bytes;

unsigned long tx_bytes;

The number of bytes received and transmitted by the interface.

unsigned long rx_errors;

unsigned long tx_errors;

The number of erroneous receptions and transmissions. There's no end of things that can go wrong with packet transmission, and the net_device_stats structure includes six counters for specific receive errors and five for transmit errors. See <linux/netdevice.h> for the full list. If possible, your driver should maintain detailed error statistics, because they can be most helpful to system administrators trying to track down a problem.

unsigned long rx_dropped;

unsigned long tx_dropped;

The number of packets dropped during reception and transmission. Packets are dropped when there's no memory available for packet data. tx_dropped is rarely used.

unsigned long collisions;

The number of collisions due to congestion on the medium.

unsigned long multicast;

17.14. Multicast

A multicast packet is a network packet meant to be received by more than one host, but not by all hosts. This functionality is obtained by assigning special hardware addresses to groups of hosts. Packets directed to one of the special addresses should be received by all the hosts in that group. In the case of Ethernet, a multicast address has the least significant bit of the first address octet set in the destination address, while every device board has that bit clear in its own hardware address.

The tricky part of dealing with host groups and hardware addresses is performed by applications and the kernel, and the interface driver doesn't need to deal with these problems.

Transmission of multicast packets is a simple problem because they look exactly like any other packets. The interface transmits them over the communication medium without looking at the destination address. It's the kernel that has to assign a correct hardware destination address; the hard_header device method, if defined, doesn't need to look in the data it arranges.

The kernel handles the job of tracking which multicast addresses are of interest at any given time. The list can change frequently, since it is a function of the applications that are running at any given time and the users' interest. It is the driver's job to accept the list of interesting multicast addresses and deliver to the kernel any packets sent to those addresses. How the driver implements the multicast list is somewhat dependent on how the underlying hardware works. Typically, hardware belongs to one of three classes, as far as multicast is concerned:

- Interfaces that cannot deal with multicast. These interfaces either receive packets directed specifically to their hardware address (plus broadcast packets) or receive every packet. They can receive multicast packets only by receiving every packet, thus, potentially overwhelming the operating system with a huge number of "uninteresting" packets. You don't usually count these interfaces as multicast capable, and the driver won't set IFF_MULTICAST in dev->flags.
- Point-to-point interfaces are a special case because they always receive every packet without performing any hardware filtering.
- Interfaces that can tell multicast packets from other packets (host-to-host or broadcast). These interfaces can be instructed to receive every multicast packet and let the software determine if the address is interesting for this host. The overhead introduced in this case is acceptable, because the number of multicast packets on a typical network is low.
- Interfaces that can perform hardware detection of multicast addresses. These interfaces can be passed a list of multicast addresses for which packets are to be received, and ignore other multicast packets. This is the optimal case for the kernel, because it doesn't waste processor time dropping "uninteresting" packets received by the interface.

The kernel tries to exploit the capabilities of high-level interfaces by supporting the third device class, which is the most versatile, at its best. Therefore, the kernel notifies the driver whenever the list of valid multicast addresses is changed, and it passes the new list to the driver so it can update the hardware filter according to the new information.

17.14.1. Kernel Support for Multicasting

Support for multicast packets is made up of several items: a device method, a data structure, and device flags:

void (*dev->set_multicast_list)(struct net_device *dev);

Device method called whenever the list of machine addresses associated with the device changes. It is also

17.15. A Few Other Details

This section covers a few other topics that may be of interest to network driver authors. In each case, we simply try to point you in the right direction. Obtaining a complete picture of the subject probably requires spending some time digging through the kernel source as well.

17.15.1. Media Independent Interface Support

Media Independent Interface (or MII) is an IEEE 802.3 standard describing how Ethernet transceivers can interface with network controllers; many products on the market conform with this interface. If you are writing a driver for an MII-compliant controller, the kernel exports a generic MII support layer that may make your life easier.

To use the generic MII layer, you should include <linux/mii.h>. You need to fill out an mii_if_info structure with information on the physical ID of the transceiver, whether full duplex is in effect, etc. Also required are two methods for the mii_if_info structure:

```
int (*mdio_read) (struct net_device *dev, int phy_id, int location);

void (*mdio_write) (struct net_device *dev, int phy_id, int location, int val);
```

As you might expect, these methods should implement communications with your specific MII interface.

The generic MII code provides a set of functions for querying and changing the operating mode of the transceiver; many of these are designed to work with the ethtool utility (described in the next section). Look in <linux/mii.h> and *drivers/net/mii.c* for the details.

17.15.2. Ethtool Support

Ethtool is a utility designed to give system administrators a great deal of control over the operation of network interfaces. With ethtool, it is possible to control various interface parameters including speed, media type, duplex operation, DMA ring setup, hardware checksumming, wake-on-LAN operation, etc., but only if ethtool is supported by the driver. Ethtool may be downloaded from http://sf.net/projects/gkernel/.

The relevant declarations for ethtool support may be found in <linux/ethtool.h>. At the core of it is a structure of type ethtool_ops, which contains a full 24 different methods for ethtool support. Most of these methods are relatively straightforward; see <linux/ethtool.h> for the details. If your driver is using the MII layer, you can use mii_ethtool_gset and mii_ethtool_sset to implement the get_settings and set_settings methods, respectively.

For ethtool to work with your device, you must place a pointer to your ethtool_ops structure in the net_device structure. The macro SET_ETHTOOL_OPS (defined in <linux/netdevice.h>) should be used for this purpose. Do note that your ethtool methods can be called even when the interface is down.

17.15.3. Netpoll

"Netpoll" is a relatively late (2.6.5) addition to the network stack; its purpose is to enable the kernel to send and receive packets in situations where the full network and I/O subsystems may not be available. It is used for features like remote network consoles and remote kernel debugging. Supporting netpoll in your driver is not, by any means, necessary, but it may make your device more useful in some situations. Supporting netpoll is also relatively easy in most cases.

Drivers implementing netpoll should implement the poll_controller method. Its job is to keep up with anything that may be happening on the controller in the absence of device interrupts. Almost all poll_controller methods take the following form:

```
void my_poll_controller(struct net_device *dev)

{

    disable_device_interrupts(dev);

    call_interrupt_handler(dev->irq, dev, NULL);
```

17.16. Quick Reference

This section provides a reference for the concepts introduced in this chapter. It also explains the role of each header file that a driver needs to include. The lists of fields in the net_device and sk_buff structures, however, are not repeated here.

#include <linux/netdevice.h>

Header that hosts the definitions of struct net_device and struct net_device_stats, and includes a few other headers that are needed by network drivers.

struct net_device *alloc_netdev(int sizeof_priv, char *name, void

(*setup)(struct net_device *);

struct net_device *alloc_etherdev(int sizeof_priv);

void free_netdev(struct net_device *dev);

Functions for allocating and freeing net_device structures.

int register_netdev(struct net_device *dev);

void unregister_netdev(struct net_device *dev);

Registers and unregisters a network device.

void *netdev_priv(struct net_device *dev);

A function that retrieves the pointer to the driver-private area of a network device structure.

struct net_device_stats;

A structure that holds device statistics.

netif_start_queue(struct net_device *dev);

netif_stop_queue(struct net_device *dev);

netif_wake_queue(struct net_device *dev);

Functions that control the passing of packets to the driver for transmission. No packets are transmitted until netif_start_queue has been called. netif_stop_queue suspends transmission, and netif_wake_queue restarts the queue and pokes the network layer to restart transmitting packets.

skb_shinfo(struct sk_buff *skb);

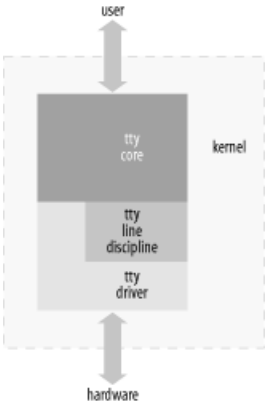
Chapter 18. TTY Drivers

A tty device gets its name from the very old abbreviation of teletypewriter and was originally associated only with the physical or virtual terminal connection to a Unix machine. Over time, the name also came to mean any serial port style device, as terminal connections could also be created over such a connection. Some examples of physical tty devices are serial ports, USB-to-serial-port converters, and some types of modems that need special processing to work properly (such as the traditional WinModem style devices). tty virtual devices support virtual consoles that are used to log into a computer, from either the keyboard, over a network connection, or through a xterm session.

The Linux tty driver core lives right below the standard character driver level and provides a range of features focused on providing an interface for terminal style devices to use. The core is responsible for controlling both the flow of data across a tty device and the format of the data. This allows tty drivers to focus on handling the data to and from the hardware, instead of worrying about how to control the interaction with user space in a consistent way. To control the flow of data, there are a number of different line disciplines that can be virtually "plugged" into any tty device. This is done by different tty line discipline drivers.

As Figure 18-1 shows, the tty core takes data from a user that is to be sent to a tty device. It then passes it to a tty line discipline driver, which then passes it to the tty driver. The tty driver converts the data into a format that can be sent to the hardware. Data being received from the tty hardware flows back up through the tty driver, into the tty line discipline driver, and into the tty core, where it can be retrieved by a user. Sometimes the tty driver communicates directly to the tty core, and the tty core sends data directly to the tty driver, but usually the tty line discipline has a chance to modify the data that is sent between the two.

Figure 18-1. tty core overview



The tty driver never sees the tty line discipline. The driver cannot communicate directly with the line discipline, nor does it realize it is even present. The driver's job is to format data that is sent to it in a manner that the hardware can understand, and receive data from the hardware. The tty line discipline's job is to format the data received from a user, or the hardware, in a specific manner. This formatting usually takes the form of a protocol conversion, such as PPP or Bluetooth.

There are three different types of tty drivers: console, serial port, and pty. The console and pty drivers have already been written and probably are the only ones needed of these types of tty drivers. This leaves any new drivers using the tty core to interact with the user and the system as serial port drivers.

To determine what kind of tty drivers are currently loaded in the kernel and what tty devices are currently present, look at the */proc/tty/drivers* file. This file consists of a list of the different tty drivers currently present, showing the name of the driver, the default node name, the major number for the driver, the range of minors used by the driver, and the type of the tty driver. The following is an example of this file:

```
/dev/tty             /dev/tty        5       0 system:/dev/tty
/dev/console         /dev/console    5       1 system:console
/dev/ptmx            /dev/ptmx       5       2 system
/dev/vc/0            /dev/vc/0       4       0 system:vtmaster
```

18.1. A Small TTY Driver

To explain how the tty core works, we create a small tty driver that can be loaded, written to and read from, and unloaded. The main data structure of any tty driver is the struct tty_driver. It it used to register and unregister a tty driver with the tty core and is described in the kernel header file <*linux/tty_driver.h*>.

To create a struct tty_driver, the function alloc_tty_driver must be called with the number of tty devices this driver supports as the paramater. This can be done with the following brief code:

```
 /* allocate the tty driver */

tiny_tty_driver = alloc_tty_driver(TINY_TTY_MINORS);

if (!tiny_tty_driver)

    return -ENOMEM;
```

After the alloc_tty_driver function is successfully called, the struct tty_driver should be initialized with the proper information based on the needs of the tty driver. This structure contains a lot of different fields, but not all of them have to be initialized in order to have a working tty driver. Here is an example that shows how to initialize the structure and sets up enough of the fields to create a working tty driver. It uses the tty_set_operations function to help copy over the set of function operations that is defined in the driver:

```
 static struct tty_operations serial_ops = {

    .open = tiny_open,

    .close = tiny_close,

    .write = tiny_write,

    .write_room = tiny_write_room,

    .set_termios = tiny_set_termios,

};



...



    /* initialize the tty driver */

    tiny_tty_driver->owner = THIS_MODULE;

    tiny_tty_driver->driver_name = "tiny_tty";

    tiny_tty_driver->name = "ttty";

    tiny_tty_driver->devfs_name = "tts/ttty%d";

    tiny_tty_driver->major = TINY_TTY_MAJOR,

    tiny_tty_driver->type = TTY_DRIVER_TYPE_SERIAL,

    tiny_tty_driver->subtype = SERIAL_TYPE_NORMAL,

    tiny_tty_driver->flags = TTY_DRIVER_REAL_RAW | TTY_DRIVER_NO_DEVFS,

    tiny_tty_driver->init_termios = tty_std_termios;

    tiny_tty_driver->init_termios.c_cflag = B9600 | CS8 | CREAD | HUPCL | CLOCAL;

    tty_set_operations(tiny_tty_driver, &serial_ops);
```

18.2. tty_driver Function Pointers

Finally, the tiny_tty driver declares four function pointers.

18.2.1. open and close

The open function is called by the tty core when a user calls open on the device node the tty driver is assigned to. The tty core calls this with a pointer to the tty_struct structure assigned to this device, and a file pointer. The open field must be set by a tty driver for it to work properly; otherwise, -ENODEV is returned to the user when open is called.

When this open function is called, the tty driver is expected to either save some data within the tty_struct variable that is passed to it, or save the data within a static array that can be referenced based on the minor number of the port. This is necessary so the tty driver knows which device is being referenced when the later close, write, and other functions are called.

The tiny_tty driver saves a pointer within the tty structure, as can be seen with the following code:

```
 static int tiny_open(struct tty_struct *tty, struct file *file)

{

    struct tiny_serial *tiny;

    struct timer_list *timer;

    int index;



    /* initialize the pointer in case something fails */

    tty->driver_data = NULL;



    /* get the serial object associated with this tty pointer */

    index = tty->index;

    tiny = tiny_table[index];

    if (tiny =  = NULL) {

        /* first time accessing this device, let's create it */

        tiny = kmalloc(sizeof(*tiny), GFP_KERNEL);

        if (!tiny)

            return -ENOMEM;



        init_MUTEX(&tiny->sem);

        tiny->open_count = 0;

        tiny->timer = NULL;



        tiny_table[index] = tiny;

    }
```

18.3. TTY Line Settings

When a user wants to change the line settings of a tty device or retrieve the current line settings, he makes one of the many different termios user-space library function calls or directly makes an ioctl call on the tty device node. The tty core converts both of these interfaces into a number of different tty driver function callbacks and ioctl calls.

18.3.1. set_termios

The majority of the termios user-space functions are translated by the library into an ioctl call to the driver node. A large number of the different tty ioctl calls are then translated by the tty core into a single set_termios function call to the tty driver. The set_termios callback needs to determine which line settings it is being asked to change, and then make those changes in the tty device. The tty driver must be able to decode all of the different settings in the termios structure and react to any needed changes. This is a complicated task, as all of the line settings are packed into the termios structure in a wide variety of ways.

The first thing that a set_termios function should do is determine whether anything actually has to be changed. This can be done with the following code:

```
 unsigned int cflag;



cflag = tty->termios->c_cflag;



/* check that they really want us to change something */

if (old_termios) {

    if ((cflag =  = old_termios->c_cflag) &&

         (RELEVANT_IFLAG(tty->termios->c_iflag) =  =

          RELEVANT_IFLAG(old_termios->c_iflag))) {

         printk(KERN_DEBUG " - nothing to change...\n");

         return;

    }

}
```

The RELEVANT_IFLAG macro is defined as:

```
 #define RELEVANT_IFLAG(iflag) ((iflag) & (IGNBRK|BRKINT|IGNPAR|PARMRK|INPCK))
```

and is used to mask off the important bits of the cflags variable. This is then compared to the old value, and see if they differ. If not, nothing needs to be changed, so we return. Note that the old_termios variable is first checked to see if it points to a valid structure first, before it is accessed. This is required, as sometimes this variable is set to NULL. trying to access a field off of a NULL pointer causes the kernel to panic.

To look at the requested byte size, the CSIZE bitmask can be used to separate out the proper bits from the cflag variable. If the size can not be determined, it is customary to default to eight data bits. This can be implemented as follows:

```
 /* get the byte size */

switch (cflag & CSIZE) {

    case CS5:

         printk(KERN_DEBUG " - data bits = 5\n");
```

18.4. ioctls

The ioctl function callback in the struct tty_driver is called by the tty core when ioctl(2) is called on the device node. If the tty driver does not know how to handle the ioctl value passed to it, it should return -ENOIOCTLCMD to try to let the tty core implement a generic version of the call.

The 2.6 kernel defines about 70 different tty ioctls that can be be sent to a tty driver. Most tty drivers do not handle all of these, but only a small subset of the more common ones. Here is a list of the more popular tty ioctls, what they mean, and how to implement them:

TIOCSERGETLSR

Gets the value of this tty device's line status register (LSR).

TIOCGSERIAL

Gets the serial line information. A caller can potentially get a lot of serial line information from the tty device all at once in this call. Some programs (such as *setserial* and *dip*) call this function to make sure that the baud rate was properly set and to get general information on what type of device the tty driver controls. The caller passes in a pointer to a large struct of type serial_struct, which the tty driver should fill up with the proper values. Here is an example of how this can be implemented:

```
 static int tiny_ioctl(struct tty_struct *tty, struct file *file,

                       unsigned int cmd, unsigned long arg)

{

    struct tiny_serial *tiny = tty->driver_data;

    if (cmd =  = TIOCGSERIAL) {

        struct serial_struct tmp;

        if (!arg)

            return -EFAULT;

        memset(&tmp, 0, sizeof(tmp));

        tmp.type        = tiny->serial.type;

        tmp.line        = tiny->serial.line;

        tmp.port        = tiny->serial.port;

        tmp.irq         = tiny->serial.irq;

        tmp.flags       = ASYNC_SKIP_TEST | ASYNC_AUTO_IRQ;

        tmp.xmit_fifo_size  = tiny->serial.xmit_fifo_size;

        tmp.baud_base       = tiny->serial.baud_base;

        tmp.close_delay     = 5*HZ;

        tmp.closing_wait    = 30*HZ;

        tmp.custom_divisor  = tiny->serial.custom_divisor;

        tmp.hub6        = tiny->serial.hub6;

        tmp.io_type     = tiny->serial.io_type;
```

18.5. proc and sysfs Handling of TTY Devices

The tty core provides a very easy way for any tty driver to maintain a file in the */proc/tty/driver* directory. If the driver defines the read_proc or write_proc functions, this file is created. Then, any read or write call on this file is sent to the driver. The formats of these functions are just like the standard /proc file-handling functions.

As an example, here is a simple implementation of the read_proc tty callback that merely prints out the number of the currently registered ports:

```
 static int tiny_read_proc(char *page, char **start, off_t off, int count,

                           int *eof, void *data)

{

    struct tiny_serial *tiny;

    off_t begin = 0;

    int length = 0;

    int i;



    length += sprintf(page, "tinyserinfo:1.0 driver:%s\n", DRIVER_VERSION);

    for (i = 0; i < TINY_TTY_MINORS && length < PAGE_SIZE; ++i) {

        tiny = tiny_table[i];

        if (tiny =  = NULL)

            continue;



        length += sprintf(page+length, "%d\n", i);

        if ((length + begin) > (off + count))

            goto done;

        if ((length + begin) < off) {

            begin += length;

            length = 0;

        }

    }

    *eof = 1;

done:

    if (off >= (length + begin))

        return 0;

    *start = page + (off-begin);

    return (count < begin+length-off) ? count : begin + length-off;

}
```

18.6. The tty_driver Structure in Detail

The tty_driver structure is used to register a tty driver with the tty core. Here is a list of all of the different fields in the structure and how they are used by the tty core:

struct module *owner;

The module owner for this driver.

int magic;

The "magic" value for this structure. Should always be set to TTY_DRIVER_MAGIC. Is initialized in the alloc_tty_driver function.

const char *driver_name;

Name of the driver, used in */proc/tty* and sysfs.

const char *name;

Node name of the driver.

int name_base;

Starting number to use when creating names for devices. This is used when the kernel creates a string representation of a specific tty device assigned to the tty driver.

short major;

Major number for the driver.

short minor_start;

Starting minor number for the driver. This is usually set to the same value as name_base. Typically, this value is set to 0.

short num;

Number of minor numbers assigned to the driver. If an entire major number range is used by the driver, this value should be set to 255. This variable is initialized in the alloc_tty_driver function.

short type;

short subtype;

Describe what kind of tty driver is being registered with the tty core. The value of subtype depends on the type. The type field can be:

TTY_DRIVER_TYPE_SYSTEM

18.7. The tty_operations Structure in Detail

The tty_operations structure contains all of the function callbacks that can be set by a tty driver and called by the tty core. Currently, all of the function pointers contained in this structure are also in the tty_driver structure, but that will be replaced soon with only an instance of this structure.

int (*open)(struct tty_struct * tty, struct file * filp);

The open function.

void (*close)(struct tty_struct * tty, struct file * filp);

The close function.

int (*write)(struct tty_struct * tty, const unsigned char *buf, int count);

The write function.

void (*put_char)(struct tty_struct *tty, unsigned char ch);

The single-character write function. This function is called by the tty core when a single character is to be written to the device. If a tty driver does not define this function, the write function is called instead when the tty core wants to send a single character.

void (*flush_chars)(struct tty_struct *tty);

void (*wait_until_sent)(struct tty_struct *tty, int timeout);

The function that flushes data to the hardware.

int (*write_room)(struct tty_struct *tty);

The function that indicates how much of the buffer is free.

int (*chars_in_buffer)(struct tty_struct *tty);

The function that indicates how much of the buffer is full of data.

int (*ioctl)(struct tty_struct *tty, struct file * file, unsigned int cmd,

unsigned long arg);

The ioctl function. This function is called by the tty core when ioctl(2) is called on the device node.

void (*set_termios)(struct tty_struct *tty, struct termios * old);

The set_termios function. This function is called by the tty core when the device's termios settings have been changed.

18.8. The tty_struct Structure in Detail

The tty_struct variable is used by the tty core to keep the current state of a specific tty port. Almost all of its fields are to be used only by the tty core, with a few exceptions. The fields that a tty driver can use are described here:

unsigned long flags;

The current state of the tty device. This is a bitfield variable and is accessed through the following macros:

TTY_THROTTLED

Set when the driver has had the throttle function called. Should not be set by a tty driver, only the tty core.

TTY_IO_ERROR

Set by the driver when it does not want any data to be read from or written to the driver. If a user program attempts to do this, it receives an -EIO error from the kernel. This is usually set as the device is shutting down.

TTY_OTHER_CLOSED

Used only by the pty driver to notify when the port has been closed.

TTY_EXCLUSIVE

Set by the tty core to indicate that a port is in exclusive mode and can only be accessed by one user at a time.

TTY_DEBUG

Not used anywhere in the kernel.

TTY_DO_WRITE_WAKEUP

If this is set, the line discipline's write_wakeup function is allowed to be called. This is usually called at the same time the wake_up_interruptible function is called by the tty driver.

TTY_PUSH

Used only internally by the default tty line discipline.

TTY_CLOSING

Used by the tty core to keep track if a port is in the process of closing at that moment in time or not.

TTY_DONT_FLIP

Used by the default tty line discipline to notify the tty core that it should not change the flip buffer when it is set.

TTY_HW_COOK_OUT

18.9. Quick Reference

This section provides a reference for the concepts introduced in this chapter. It also explains the role of each header file that a tty driver needs to include. The lists of fields in the tty_driver and tty_device structures, however, are not repeated here.

#include <linux/tty_driver.h>

Header file that contains the definition of struct tty_driver and declares some of the different flags used in this structure.

#include <linux/tty.h>

Header file that contains the definition of struct tty_struct and a number of different macros to access the individual values of the struct termios fields easily. It also contains the function declarations of the tty driver core.

#include <linux/tty_flip.h>

Header file that contains some tty flip buffer inline functions that make it easier to manipulate the flip buffer structures.

#include <asm/termios.h>

Header file that contains the definition of struct termio for the specific hardware platform the kernel is built for.

struct tty_driver *alloc_tty_driver(int lines);

Function that creates a struct tty_driver that can be later passed to the tty_register_driver and tty_unregister_driver functions.

void put_tty_driver(struct tty_driver *driver);

Function that cleans up a struct tty_driver structure that has not been successfully registered with the tty core.

void tty_set_operations(struct tty_driver *driver, struct tty_operations *op);

Function that initializes the function callbacks of a struct tty_driver. This is necessary to call before tty_register_driver can be called.

int tty_register_driver(struct tty_driver *driver);

int tty_unregister_driver(struct tty_driver *driver);

Functions that register and unregister a tty driver from the tty core.

*void tty_register_device(struct tty_driver *driver, unsigned minor, struct*

*device *device);*

Chapter 19. Bibliography

Most of the information in this book has been extracted from the kernel sources, which are the best documentation about the Linux kernel.

Kernel sources can be retrieved from hundreds of FTP sites around the world, so we won't list them here.

Version dependencies are best checked by looking at the patches, which are available from the same places where you get the whole source. The program called repatch might help you in checking how a single file has been modified throughout the different kernel patches; it is available in the source files provided on the O'Reilly FTP site.

19.1. Books

While the bookstores are full of technical books, there are surprisingly few that are directly relevant to Linux kernel programming. Here is a selection of books found on our shelves.

19.1.1. Linux Kernel

Bovet, Daniel P. and Marco Cesate. Understanding the Linux Kernel, Second Edition. Sebastopol, CA: O'Reilly & Associates, Inc. 2003.

This book covers the design and implementation of the Linux kernel in great detail. It is more oriented toward providing an understanding of the algorithms used than documenting the kernel API. This book covers the 2.4 kernel but still contains a great deal of useful information.

Gorman, Mel. Understanding the Linux Virtual Memory Manager. Upper Saddle River, NJ: Prentice Hall PTR, 2004.

Developers wanting to know more about the Linux virtual memory subsystem may wish to have a look at this book. It is centered around the 2.4 kernel but contains 2.6 information as well.

Love, Robert. Linux Kernel Development. Indianapolis: Sams Publishing, 2004.

This book covers Linux kernel programming with a broad scope. It is a reference that should be on every Linux hacker's bookshelf.

Yaghmour, Karim. Building Embedded Systems. Sebastopol, CA: O'Reilly & Associates, Inc. 2003.

This book will be useful to those writing Linux code for embedded systems.

19.1.2. Unix Design and Internals

Bach, Maurice. The Design of the Unix Operating System. Upper Saddle River, NJ: Prentice Hall, 1987.

Though quite old, this book covers all the issues related to Unix implementations. It was the main source of inspiration for Linus in the first Linux version.

Stevens, Richard. Advanced Programming in the UNIX Environment. Boston: Addison-Wesley, 1992.

Every detail of Unix system calls is described herein, which is a good companion when implementing advanced features in the device methods.

Stevens, Richard. Unix Network Programming. Upper Saddle River, NJ: Prentice Hall PTR, 1990.

Perhaps the definitive book on the Unix network programming API.

19.2. Web Sites

In the fast-moving world of Linux kernel development, the most current information is often found online. The following is our selection of the best web sites as of this writing:

http://www.kernel.org

ftp://ftp.kernel.org

This site is the home of Linux kernel development. You'll find the latest kernel release and related information. Note that the FTP site is mirrored throughout the world, so you'll most likely find a mirror near you.

http://www.bkbits.net

This site hosts the source repositories used by a number of prominent kernel developers. In particular, the project called "linus" contains the mainline kernel as maintained by Linus Torvalds. If you are curious about the very latest patches which have been applied to the kernel, this is the place to look.

http://www.tldp.org

The Linux Documentation Project carries a lot of interesting documents called "HOWTOs"; some of them are pretty technical and cover kernel-related topics.

http://www.linux.it/kerneldocs

This page contains many kernel-oriented magazine articles written by Alessandro Rubini. Some of them date back a few years, but they usually still apply; some of them are in Italian, but usually an English translation is available as well.

http://lwn.net

At the risk of seeming self-serving, we point out this news site that, among other things, offers regular kernel development coverage and API change information.

http://www.kerneltraffic.org

Kernel Traffic is a popular site that provides weekly summaries of discussions on the Linux kernel development mailing list.

http://www.kerneltrap.org/

This site picks up occasional interesting developments in the Linux and BSD kernel communities.

http://www.kernelnewbies.org

This site is oriented toward new kernel developers. There is beginning information, a FAQ, and an associated IRC channel for those looking for immediate assistance.

http://janitor.kernelnewbies.org/

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index