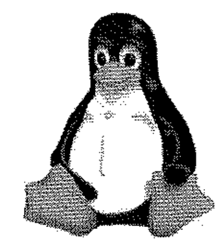
Writing

Linux Device Drivers

a guide with exercises



Including the 2.6.31 kernel

Jerry Cooperstein

Published by:

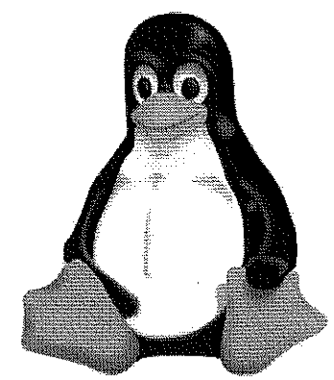
Jerry Cooperstein
coop@coopj.com
www.coopj.com

Contents

List of Figures

List of Tables

Preface



Objectives

Writing Linux Device Drivers is designed to show experienced programmers how to develop device drivers for **Linux** systems, and give them a basic understanding and familiarity with the **Linux** kernel.

Upon mastering this material, you will be familiar with the different kinds of device drivers used under **Linux**, and know the appropriate **API**'s through which devices (both hard and soft) interface with the kernel.

We will focus primarily on **Device Drivers** and only secondarily on the **Linux Kernel**. These are impossible to separate, since device drivers are an integral part of the kernel. However, most device drivers use only a limited set of kernel functions and one need not learn everything about the kernel to do a device driver. Yet while device drivers don't control important kernel features such as scheduling or memory management, the more you know about how **Linux** handles such things the better a device driver you can write.

In many other operating systems, which are closed source, there is a cleaner separation between a device driver and the kernel proper. Because **Linux is open source**, the device driver developer has full access to all of the kernel. This is both powerful and dangerous.

While we will discuss kernel internals and algorithms we will examine deeply only the functions which are normally used in device drivers. More details on things such as scheduling, memory management, etc., belong more properly in a higher level treatment (or lower level depending on how you define things.)

Developing device drivers is a big subject both in depth (from deep inside the kernel to usage in user-space) and in breadth (the many types of devices.) In order to keep things manageable we are going to limit our range both vertically and horizontally.

This means we won't look very deeply into the kernel's inner plumbing even as it relates to device drivers. And for particular types of device drivers we will stop before we get to detailed aspects of particular devices or classes of devices and hardware. It also means we are going to just ignore whole classes of devices, such as **SCSI** and **wireless**, as any treatment of these subjects would rapidly become both huge and specialized.

Our order of presentation is not axiomatic; i.e., we will have some forward referencing and digressions. The purpose is to get you into coding as quickly as possible. Thus we'll tell you early on how to dynamically allocate memory in the simplest way, so you can actually write code, and then later cover the subject more thoroughly. Furthermore, the order of subjects is flexible, so feel free to vary it according to your interests.

Who You Are

You are interested in learning how to write device drivers for the **Linux** operating system. Maybe you are just doing this for fun, but more likely you have this task as part of your job. The purpose here is to ease your path and perhaps shorten the amount of time it takes to reach a level of basic competence in this endeavor.

How much you get out of this and how bug-free, efficient, and optimized your drivers will be depends on how good a programmer you were before you started with the present material. There is no intent or time here to teach you the elements of good programming or the **Linux** operating system design in great detail.

There are two reasons for this disclaimer:

- First, there is no shortage of books and classes on programming methods. For the **Linux** kernel the choices are fewer, but they exist and we will mention some of them at appropriate times.
- Second, my knowledge is not as deep as those who have contributed greatly to the development of **Linux**, and programming is not my strength. Indeed my personal contributions to the kernel code base have been very minor. I'll explain shortly why I've produced this material despite these facts.

You should:

- Be proficient in the **C** programming language.
- Be familiar with basic **Linux** (**Unix**) utilities, such as **ls, rm, grep, tar**, and have a familiarity with command shells and scripts.
- Be comfortable using any of the available text editors (e.g., **vi, emacs.**)
- Know the basics of compiling and linking programs, constructing Makefiles etc.; i.e., be comfortable doing application developing in a **Linux** or **Unix** environment.
- Have a good understanding of systems programming in a **Unix** or **Linux** environment, at least from the standpoint of writing applications.

- Experience with any major **Linux** distribution is helpful but is not strictly required.

If you have had some experience configuring and compiling kernels, and writing kernel modules and or device drivers, you will get much more out of this material.

If you have a good grasp of operating system fundamentals and familiarity with the insides of any other operating system, you will gain much more from this material.

While our material will not be very advanced, it will strive to be thorough and complete. It is worth repeating that we are not aiming for an expert audience, but instead for a competent and motivated one.

My History and Why I'm Doing This.

By training I'm a nuclear physicist; I have a PhD in theoretical nuclear astrophysics and I did research on dense nuclear matter, supernova explosions, neutrino diffusion, hydrodynamics, shockwaves, general relativity, etc. for a couple of decades and published dozens of papers, review articles and book chapters in the main physics and astrophysics journals. I was on the faculty at a number of major universities and government labs.

I've been teaching in one form or another for more than 30 years. I've taught advanced as well as introductory courses on a wide variety of subjects in physics and astrophysics at both the undergraduate and graduate level as well as supervised a good number of students. And I've been teaching material such as the present subject matter for more than a decade.

So you may be asking where I get the nerve to prepare a book of this nature? The answer is I stumbled into it. I've never worked primarily as a software or hardware engineer and my path to **Linux** was very non-linear.

I've used and programmed for computers for a long time. The first time I sat down at a computer was 1969; the machine was a DEC PDP-9 and the keyboard was an actual teletype with booting done through paper tapes. I've used every operating system and programming language that has been thrown my way over these four decades and fortunately I've been able to forget about most of them as they became deprecated or obsolete.

Except for some low-level data acquisition software in my early student days, all my computer-related work during my career as a physicist was at the application level. My main projects involved the large scale numerical simulations of exploding stars, together with the numerical data and graphical analysis code that was required.

In 1994 I left academia and entered the business world. I spent the next 5 years working as a consultant with a major petrochemical company, helping with the geophysics and seismic analysis software used for oil exploration and recovery. (Equations are equations whether they describe colliding nuclei, exploding stars, or seismic waves propagating through layers of the earth.)

During this period I began to use **Linux** extensively as it was a **Unix**-like platform which I could use to develop and debug code which would then be run on large supercomputer platforms. Towards the end of this period I was advocating moving to **Linux** clusters as a better way to get bang for the buck.

In 1998 my oil company was devoured by a larger one and its technology division was decimated. Oil was selling for less than $12 a barrel at the time and the corporate wisdom was that research and development was not very important.

At the same time a major chips manufacturer approached my employer and asked us to develop materials to train a bunch of **NT** engineers to work with **Linux**. We found this amusing since a couple of years before they has asked us to train **Unix** engineers to work on **NT**. I was tasked with developing materials and teaching from them and I have been doing it ever since.

Eventually this project grew into three main classes. One was on systems programming, a second was on **Linux** kernel internals, and the third was on **Linux** device drivers. My company, **Axian** of Beaverton, OR, funded and deployed these classes. Eventually we franchised the material out, with some modifications, to be used by various **Linux** distributors. In particular all three classes were used by **Red Hat Inc** for about 10 years as their curriculum for **Linux** developers.

Over that same period I also did a number of engineering projects, mostly involving device drivers or developing specialized embedded **Linux** platforms. But I spent the lion's share of my time teaching and preparing courseware.

Over those ten years I personally taught sessions of these classes at least 100 times and also functioned as the courseware maintainer and contact person for the many other instructors who taught from this material world-wide, and who contributed in a major way to its improvement.

Great efforts were made to keep the material up to date, since the **Linux** kernel morphs rather quickly. New editions were published four times a year, gradually coming into sync with the new kernel release schedule.

Over the years many students had requested that the courseware material be obtainable in bookstores or mail order. Because it was not possible to publish the courseware because of contractual requirements, it was only possible to obtain it by enrolling in a class, a relatively expensive proposition.

In 2009 the **Axian-Red Hat** contract expired and simultaneously I left **Axian**'s staff. I decided (with **Axian**'s generous permission) to find a way to rework the material and publish it.

And that is how we got here. This it not just a repackaging of the courses that were previously marketed and delivered. There has been a rather major rewrite, development of new exercises, addition of new material and deletion of old material.

Even though I have been involved in **Linux** for more than 15 years I am an outsider. My contributions to the kernel source are not worth noting, I don't know personally major kernel developers, I don't go to conferences, and other than some email relationships with some well-known folks that arose out of my doing technical review on a number of their **Linux**-kernel related books, I'm pretty much an observer and consumer.

I hope that makes my perspective useful to you if you are new to this field, as there is a lot of **Linux**-related chatter, news and documentation which assumes more familiarity than it should, or thinks things are more obvious or easier than they are to the non-expert. In preparing teaching materials over the years I've often had to do a lot of hard work to first understand and then present in a simple way things that were assumed to be obvious.

Linux Developer Classes Now Available

I don't expect to get rich by publishing this material. I do hope that if you have **Linux** programming training needs you view it as a good advertisement for engaging our services for live in-person training classes.

The following classes are available:

- Linux Systems Programming
- Linux Device Drivers
- Linux Kernel Internals

Each of these classes are a full five days in length. Customization and combination options are available. Until demand gets out of control your author is expected to be the instructor.

For detailed descriptions and outlines and pricing and logistics visit **http://www.coopj.com** and contact **coop@coopj.com**.

Acknowledgments

First of all I must thank my employer of over 15 years, Axian Inc (**http://www.axian.com**) of Beaverton, Oregon, for giving me permission to use material originally under Axian copyright and which was developed on its dime. In particular, Frank Helle and Steve Bissel have not only been extremely generous in allowing me these rights, but have been true friends and supporters in everything I've done.

In the more than a decade I supervised **Linux** developer classes for Axian (which were most often delivered through **Red Hat**'s training division), I interacted with a large number of instructors who taught from the material I was responsible for. They made many suggestions, fixed errors and in some cases contributed exercises. Colleagues I would like to express a very strong thank you to include Marc Curry, Dominic Duval, Terry Griffin, George Hacker, Tatsuo Kawasaki, Richard Keech, and Bill Kerr.

I would also like to thank Alessandro Rubini for his warm and generous hospitality when not long after I began teaching about device drivers and **Linux**, I showed up at his home with my whole family. I also thank him for introducing me to the kind folks at O'Reilly publishing who gave me the opportunity to help with the review of their **Linux** kernel books, which has expanded my knowledge enormously and introduced me to a number of key personalities.

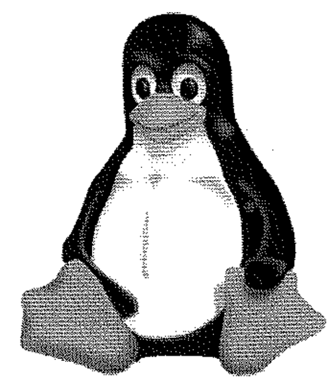
The biggest acknowledgment I must give is to the students who have contributed to the material by asking questions, exposing weaknesses, requesting new material and furnishing their real life experiences and needs, which has hopefully kept the material from being pedantic and made it more useful. Without them (and the money they paid to sit in classes and be forced to listen to and interact with me) this presentation would not exist.

I must also thank my family for putting up me with through all of this, especially with my frequent travels.

Finally, I would like to acknowledge the late Hans A. Bethe, who taught me to never be frightened of taking on a task just because other people had more experience on it.

Chapter 1

Preliminaries



We'll discuss our procedures. We'll also make some comments about **Linux** distributions, kernel versions and hardware platforms. We'll promote the **Linux Driver Project**. Finally we'll point to some sources of documentation.

1.1 Procedures

You will need a computer installed with a current **Linux** distribution, with the important developer tools (for compiling, etc.) properly deployed.

The emphasis will be on hands-on programming, with most sections having laboratory exercises. Where feasible labs will build upon previous lab assignments. The solution set can be retrieved from

http://www.coopj.com/LDD. As they become available, errata and updated solutions will also be posted on that site.

Lab **solutions** are made available so you can see at least one successful implementation, and have a possible template to begin the next lab exercise if it is a follow up. In addition, **examples** as shown during the exposition are made available as part of the `SOLUTIONS` package, in the `EXAMPLES` subdirectory. Once you have obtained the solutions you can unpack it with:

```
tar zxvf LDD*SOLUTIONS*.tar.gz
```

substituting the actual name of the file.

In the main solutions directory, there is a **Makefile** which will recursively compile all subdirectories. It is smart enough to differentiate between kernel code, user applications, and whether multi-threading is used.

There are some tunable features; by default all sub-directories are recursively compiled against the source of the currently running kernel. One can narrow the choice of directories, or use a different kernel source as in the following examples, or even pick a different architecture:

```
make SDIRS=s_22
make KROOT=/lib/modules/2.6.31/build
make SDIRS="s_0* s_23" KROOT=/usr/src/linux-2.6.31/build
make ARCH=i386
```

where `KROOT` points to the kernel source files. On an **x86_64** platform, specifying `ARCH=i386` will compile 32-bit modules. The **genmake** script in the main directory is very useful for automatically generating makefiles, and is worth a perusal.

For this to work, the kernel source has to be suitably prepared; in particular it has to have a **configuration file** (`.config` in the main kernel source directory) and proper dependencies set up.

One should note that we have emphasized clarity and brevity over rigor in the solutions; e.g., we haven't tried to catch every possible error or take into account every possible kernel configuration option. The code is not bullet-proof; it is meant to be of pedagogical use.

If you have any questions or feedback on this material contact us at ***coop@coopj.com***.

- The provided solutions will from time to time contain functions and features not discussed in the main text.
- This is done to illustrate methods to do more than the minimum work to solve the problem and teach extra material.
- If there is anything that **must** be used and is not covered in the material, its omission is a bug, not a feature, and should be brought to our attention.

1.2 Linux Distributions

There are many **Linux** distributions, ranging from very widely used to obscure. They vary by intended usage, hardware and audience, as well as support level. A very comprehensive list can be found at **http://lwn.net/Distributions/**.

We have tried to keep this material as distribution-agnostic as possible and thus we will focus on **vanilla** kernels as obtained from **http://www.kernel.org**. For all but the most specialized distributions this won't present any inconveniences.

You should be able to do any of the laboratory exercises provided herein on any major distribution using the vendor-supplied kernel, as long as you have the kernel source or development packages installed. Furthermore, the supplied solutions should compile and run as long as the kernel is not too antique; some minor twiddling may be necessary for kernels earlier than about 2.6.26.

Occasionally which distribution you are using will matter, but this should only happen when we (reluctantly) descend into system administration, such as when we must describe the location of particular files and directories or how to install certain required software packages. Fortunately, modern distributions differ much less in these matters than they did in the early days of **Linux** and we will rarely have to deal with such inconveniences.

The material has been developed primarily on **Red Hat**-based systems, mostly on 64-bit variants with testing also done on 32-bit systems. But it has also been tested on a number of other distributions. Explicitly we have used:

- **Red Hat Enterprise Linux 5.3**
- **Fedora 11**
- **Centos 5.3**
- **Scientific Linux 5.3**
- **Open Suse 11.1**
- **Debian Lenny**
- **Ubuntu 9.04**
- **Gentoo**

As far as software installation and control is concerned distributions tend to use either **RPM**-based or **deb**-based package management. In the above list **Red Hat**, **Fedora**, **Centos**, **Scientific Linux** and **OpenSUSE** are **RPM**-based, and **Debian** and **Ubuntu** are **deb**-based. When necessary we will give required instructions for either of these two broad families.

GENTOO is based on neither of this packaging systems, and instead uses the **portage/emerge** system which involves compiling directly from source. If you are a **GENTOO** user, and you have successfully accomplished a fully functional installation (which is generally not a task for novices) you won't need detailed instructions in how to install software or find things, and so we won't insult you by offering it.

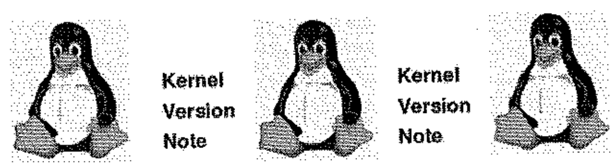
If you are running any other distribution you shouldn't have any trouble adapting what we are doing to your installation.

1.3 Kernel Versions

For this class we have a number of choices to use for our running kernel and kernel sources. We will be working with the latest stable **Linux** kernel series, version **2.6**.

The lab exercise solutions are designed to work with the most recent **Linux** kernels (2.6.31 as of this writing) and with some minor tweaks (which are incorporated in the solutions) as far back as 2.6.26 and even earlier (2.6.18) with a few more tweaks.. They should also work with all major distributor kernels of the same vintage.

Because the **2.4** kernel series is still occasionally in usage, we will point out important features that have changed. However, differences with the previous generations of production kernels (**2.2, 2.0**) will be mentioned only briefly for historical purposes.



- From place to place in this material you will find boxes like these.
- They may highlight features appearing in the **2.6** kernel that are new or modified, under active development, or marked as deprecated.
- They may also highlight features in use in the **2.4** kernel that have become obsolete and should not be used in new code.

The parts of the source to the **Linux** kernel that are necessary for external module compilation (the **kernel-devel** package) should already be installed on your system.

The latest **stable** official releases of the various kernel versions can always be obtained from **http://www.kernel.org**, as can the latest development kernel, released by **Linus Torvalds**. A number of ***funneling*** source trees also exist, but are not given the same prominence at the main kernel web site. For examples there are staging trees for network issues, **USB**, etc.

One more tree deserving special mention is the **linux-next** project, information about which is at **http://linux.f-seidel.de/linux-next/pmwiki/**. This kernel tree accumulates patch sets for the **next** kernel version; i.e., while 2.6.31 is being developed the **linux-next** tree collects and debugs material appropriate for 2.6.32 and shakes out conflicts etc.

By using the site's **finger** server, you can ascertain the latest kernel versions:

```
$ finger @kernel.org
```

```
The latest stable version of the Linux kernel is:              2.6.30.5
The latest prepatch for the stable Linux kernel tree is:       2.6.31-rc7
The latest snapshot for the stable Linux kernel tree is:       2.6.31-rc7-git2
The latest 2.4 version of the Linux kernel is:                 2.4.37.5
```



```
The latest 2.2 version of the Linux kernel is:                 2.2.26
The latest prepatch for the 2.2 Linux kernel tree is:          2.2.27-rc2
The latest -mm patch to the stable Linux kernels is:           2.6.28-rc2-mm1
```

Kernel version numbering is done with 3 integers, the first two indicating the major release number, and the third the minor release number; i.e., 2.6.31 or 2.4.33. In addition an extra string may be appended for identification purposes, but is not truly part of the kernel version number. For instance, if a kernel is named **2.6.29.2-rt10**, we have

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 29
EXTRAVERSION = .2-rt10
```

in the main **Makefile** in the kernel source.

The file **/usr/src/linux/include/linux/version.h** provides some macros to make life easier:

```
#define LINUX_VERSION_CODE 132638
#define KERNEL_VERSION(a,b,c) (((a) << 16) + ((b) << 8) + (c))
```

where LINUX_VERSION_CODE is obtained using the KERNEL_VERSION() macro. Note an equivalent statement would be:

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

This header file is constructed during compilation, and won't exist on a pristine source.

Version-dependent code can be handled as in this example:

```
#include <linux/version.h>
....
#if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,0)
    call_usermodehelper(cmd,argv,envp);
#else
    call_usermodehelper(cmd,argv,envp,wait);
#endif
```

- Such explicit version dependence is frowned up on in the official kernel tree. One is supposed to abandon backwards compatibility for earlier kernel versions so as to keep things uncluttered and most efficient.
- This attitude is often viewed as inconvenient by external device driver maintainers as it makes it necessary to maintain different driver sources for older kernels if bugs or security holes are patched.

Beginning with the 2.6.11 kernel, a new scheme was adopted in which a fourth digit is used for small patches that concern serious bugs and security matters. Only non-controversial patches are accepted, and are numbered as in 2.6.11.1, 2.6.11.2, 2.6.11.3 etc.

The idea has brought a good measure of stability to the development process and quieted some complaints about not having a stable enough kernel for production. Rules concerning what sort of patches are accepted for this **stable** kernel tree can be found at **/usr/src/linux/Documentation /stable_kernel_rules.txt**.

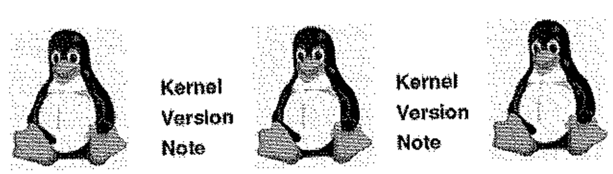
1.4 Platforms

We will try and be as platform-independent as possible. However, we will consider the **x86** architecture explicitly from time to time and we won't concentrate on **Linux** on **IA64, Alpha, SPARC, PPC, ARM**, etc., very much.

There are at least three reasons to consider the **x86** architecture explicitly:

- There are parts of the **Linux** kernel which must be platform-dependent, and to understand what is going on some specificity will be needed.
- You are probably working on an are **x86** machine right now..
- Most people developing for **Linux** are concentrating on this platform.

We'll try to make any platform-dependence explicit. All solutions have been tested on 32-bit and 64-bit **x86** architectures.



- The 2.6.24 kernel contained a merge of the 32-bit **i386** and 64-bit **x86_64** architectures. The new name is simply **x86**.
- This involved a major reorganization and rewrite of code and project completion has taken a while.
- Many files were reorganized; for instance instead of `arch/i386/mm/init.c` we have `arch/x86/mm/init_32.c` and `arch/x86/mm/init_64.c`.

1.5 Hardware

Sometimes when people teach device drivers they use simple devices hanging off an external port. Rather than do this we will use the hardware already on the machine such as network cards and input devices, and piggyback our device drivers on top of the already installed ones using the kernel's ability to share interrupts.

Questions often come up of the following variety:

- How many I/O ports does my device use, and what addresses do they use?
- What IRQ?
- Do I read bytes or words, how many per interrupt, etc?
- What standards does the device conform to?

These questions can be answered **only** from the hardware's specifications, and sometimes by examining the hardware itself. When you are writing a device driver you **must** have such knowledge and if you don't you can't write a driver. (Of course it is possible to figure out a lot by probing a device which keeps its specifications secret, and a lot of drivers have been reverse engineered this way. But as **Linux** has matured this has become much rarer and time and energy are better spent encouraging manufacturers to cooperate if they want their devices supported, than in doing this kind of dirty work.)

Get as much information as you can from the hardware people, but be prepared for some of it to be wrong or out of date, especially with new devices. It is not unusual for the hardware and the specifications to not be in sync or for a device to fail to completely follow specifications. Sometimes this is because device manufacturers are content with making sure the device works adequately under the market-dominant operating system and then stop asking questions at that point.

1.6 Linux Driver Project

The **Linux Driver Project** is a group of kernel developers and some project managers that provides open source device drivers free of charge. The project will work closely with manufacturers and when necessary will sign **NDAs** (Non-Disclosure Agreements) as long as the final work has a proper **GPLv2** license.

The project is headed by Greg Kroah-Hartman whose weblog is at **http://www.kroah.com/log**. New volunteers are always welcome. Its main web page is at **http://linuxdriverproject.org /twiki/bin/view**.

A goal of the project is to bring many external drivers into the main kernel tree. A list of potential candidates is kept at **http://linuxdriverproject.org/twiki/bin/view/Main/OutOfTreeDrivers**. Also maintained is a list of devices for which no **Linux** drivers exist, which can be found at **http://linuxdriverproject.org/twiki/bin/view/Main/DriversNeeded**.

1.7 Documentation and Links

The **best** source of documentation about the **Linux** kernel is the source itself. In many cases it is the **only** documentation. Never trust what you see in books (including this one) or articles without looking at the source.

The **/usr/src/linux/Documentation** directory contains a many useful items. Some of the documentation is produced using the **docbook** system (see **http://www.docbook.org**.) To produce this you go to `/usr/src/linux` and type

```
make { htmldocs | psdocs | pdfdocs | rtfdocs }
```

the different forms giving you the documentation in either as web-browseable, postscript, portable document format, or rich text format, which will appear in the **/usr/src/linux/Documentation /DocBook** directory. Warning: producing this documentation takes longer that compiling the kernel itself! For this to work properly you may have to install additional software on your system, such as **jade** or **latex**.

Books

Linux Device Drivers, Third Edition, by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, pub. O'Reilly, 2005.

The full, unabridged on-line version can be viewed at **http://lwn.net/Kernel/LDD3/** and downloaded from **http://lwn.net/Kernel/LDD3/ldd3_pdf.tar.bz2**. and source code for the examples in the book can be retrieved at **http://examples.oreilly.com/linuxdrive3/examples.tar.gz**.

Understanding the Linux Kernel, Third Edition, by Daniel P. Bovet and Marco Cesati, pub. O'Reilly, 2005.

Understanding Linux Network Internals, by Christian Benvenuti, pub. O'Reilly, 2006.

Linux Kernel in a Nutshell, by Greg Kroah-Hartman, pub. O'Reilly, 2006.

The full text of the book can be viewed or downloaded at **http://www.kroah.com/lkn/**.

(**Disclaimer:** The author was a technical reviewer on various editions of the previous four O'Reilly books.)

Linux Kernel Development, Second Edition, by Robert Love, pub. Novell Press, 2005.

Linux Debugging and Performance Tuning: Tips and Techniques, by Steve Best, pub. Prentice Hall, 2005.

Kernel Development and Mailing List Sites

http://lwn.net
Linux Weekly News: Latest **Linux** news including a Kernel section. This very important site is supported by user subscriptions, so please consider making an individual or corporate contribution!

http://ldn.linuxfoundation.org/book/how-participate-linux-community
A complete view of the kernel development process and how to join it.

http://lwn.net/Articles/driver-porting
A compendium of Jonathan Corbet's articles on porting device drivers to the 2.6 kernel.

http://www.linux-foundation.org/en/Linux_Weather_Forecast
Tracks ongoing kernel developments that are likely to achieve incorporation in the near future.

http://lkml.org/
Archive of the Kernel Mailing List, updated in real time.

http://www.tux.org/lkml
The Kernel Mailing List **FAQ:** How to subscribe, post, etc. to the kernel mailing list, plus related matters such as how to submit and use patches.

http://www.zip.com.au/ akpm/linux/patches/stuff/tpp.txt
Andrew Morton's guide to the **perfect patch.**

http://linux.yyz.us/patch-format.html
A detailed guide for how to submit patches to the official kernel tree.

http://linux.yyz.us/git-howto.html
The **Kernel Hackers' Guide to git**, the source code management system, used by many senior kernel developers.

http://lxr.linux.no
The **lxr** kernel browser: Can be run though the Internet or installed locally.

http://www.kernelnewbies.org
kernelnewbies: An excellent source of documentation; while starting at the lowest level and going to advanced.

http://www.kernelnewbies.org/LinuxChanges
Comprehensive kernel changelog: A detailed list of changes in the kernel and its **API** from one release to another. .

http://www.kerneltrap.org
kerneltrap: Contains interviews, white papers, etc.

Other Documentation

http://www2.axian.com/ coop/linux_pubs/
Several articles and talks by Jerry Cooperstein about changes for the 2.6 kernel.

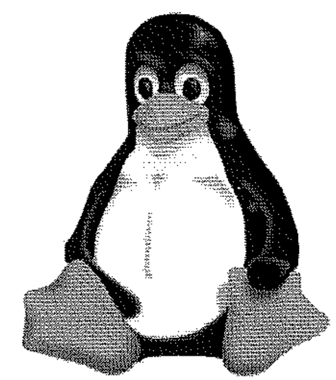
http://www.linuxsymposium.org/2007/archives.php/
http://ols.fedoraproject.org/OLS/
Full proceedings of Ottawa **Linux** Symposium from 2001 on, containing many important talks and papers.

http://www.ibm.com/developerworks/linux
IBM's Linux developer page, with white papers and other documentation.

http://www.tldp.org
contains a lot of material from the **Linux Documentation Project (LDP)**, including all current **HOWTO** documents.

Chapter 2

Device Drivers



We'll discuss how device drivers are used and consider the different types of devices; i.e., character, block and network. We'll discuss the difference between mechanism and policy. We'll consider the disadvantages of loading binary blobs. We will then take a quick tour of how applications interface with device drivers and make system calls. We'll see how errors are defined, and how to obtain kernel output using **printk()**. We'll consider all this in detail later.

2.1 Types of Devices

A **device driver** is the lowest level of software as it is directly bound to the hardware features of the device. The kernel can be considered an application running on top of device drivers; each driver manages one or more piece of hardware while the kernel handles process scheduling, filesystem access, interrupts, etc.

Drivers may be integrated directly into the kernel, or can be designed as loadable **modules**. Not all modules are device drivers. A driver can be designed as either a modular or a built-in part of the kernel, with little or no change of the source.

In the usual device taxonomy there are three main types:

Character Devices

- Can be written to and read from a byte at a time.
- Well represented as **streams**.
- Usually permit only sequential access.
- Can be considered as files.
- Implement **open, close, read,** and **write** functions.
- Serial/parallel ports, console (monitor and keyboard), etc.
- Examples: `/dev/tty0, /dev/ttyS0, /dev/dsp0, /dev/lp1`

Block Devices

- Can be written to and read from only in block-size multiples; access is usually **cached**.
- Permit random access.
- Filesystems can be mounted on these devices.
- In **Linux** block devices can behave like character devices, transferring any number of bytes at a time.
- Hard drives, cdroms, etc.
- Examples: `/dev/hda1, /dev/fd0`

Network Devices

- Transfer packets of data. Device sees the packets, not the streams.
- Most often accessed via the BSD socket interface.
- Instead of **read, write,** the kernel calls packet reception and transmission functions.
- Network interfaces are not mapped to the filesystem; they are identified by a name.
- Examples: `eth0, ppp0`

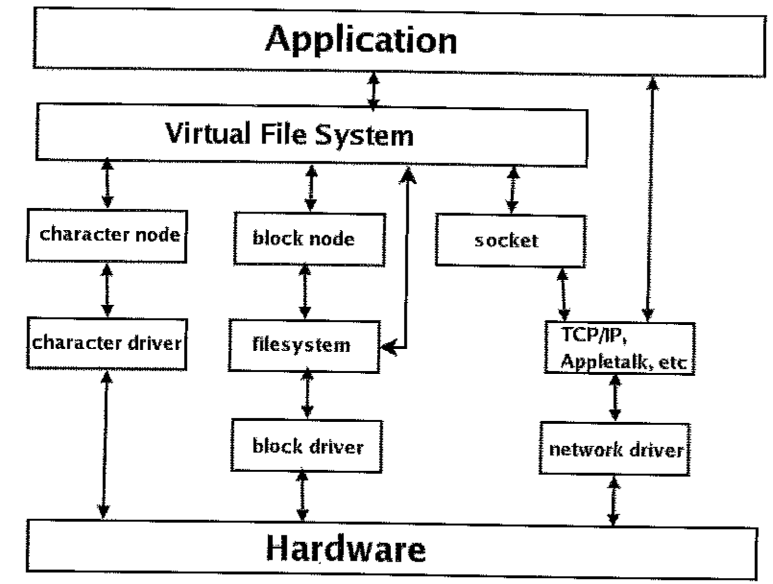


Figure 2.1: **From application to device**

Device Types and User-Hardware Connection

What differentiates the types of drivers is the methods they use to connect the kernel with user-space. Most of the time the connection passes through the **VFS** (Virtual File System), and then what methods are invoked depends on whether the access is to a character device node, block device node, or a socket.

Character drivers may or may not work on character streams; the essential thing is they are most directly connected to the user and to the hardware.

Block drivers are connected to the hardware, and to the user through the filesystem, caching, and the Virtual File System (VFS).

Network drivers are connected to the hardware and to the user through various kinds of protocol stacks.

Other Types of Devices

There are other types of devices which don't fit precisely in the character/block/network division (although functionally they can be used for any of these three generic classes of peripherals.)

SCSI (Small Computers Systems Interconnect) devices share an underlying protocol regardless of function. The hard work goes into writing the driver for the controller hardware which may run many devices.

USB (Universal Serial Bus) devices also share an underlying protocol. Once again there is a lower layer of drivers tied to the controller hardware, and then device-specific drivers for the various peripherals connected to the bus.

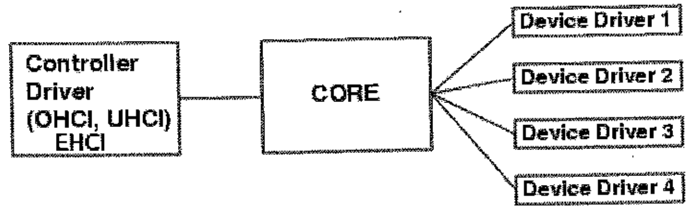


Figure 2.2: **USB: Controller, Core and Device**

User-space drivers, such as the video drivers incorporated into **X.Org** (**http://www.x.org**), work completely in user-space, but are given privileges to directly address hardware. They use the **ioperm()** and **iopl()** functions to accomplish this, which we'll discuss when we discuss reading and writing to I/O ports.

2.2 Mechanism vs. Policy

Device drivers should maintain a clear distinction between **mechanism** and **policy**.

By **mechanism**, we mean providing flexibly the abilities that the device itself can capably perform. By **policy**, we mean controlling how those capabilities are used.

In other words it is not up to the driver to enforce certain decisions (unless there is a hardware limitation) such as:

- How many processes can use the device at once.
- Whether I/O is blocking or non-blocking, synchronous or asynchronous, etc.
- Whether certain combinations of parameters can never occur even if they are unwise.

Often a driver may come with a user-space control program, or daemon, which has the capability of controlling device policy. As such, it should provide methods of setting parameters and modifying behaviour, perhaps through the use of **ioctl**'s, **/proc**, **/sys** etc.

A driver which fails to distinguish between mechanism and policy is a driver destined for trouble. Tomorrow's user may have quite different needs than today's. Being human, the driver developer may even forget why a narrowing of choices was made. One should never underestimate the likelihood of a user behaving in an unexpected fashion.

2.3 Avoiding Binary Blobs

There is a method of deploying a **Linux** device driver which has been promoted by certain vendors, with which we strongly disagree.

The essence of this method is two separate the driver into two parts:

- A binary blob, for which the source is not given. This blob may contain all or part of the driver from another operating system (usually from you-know-who.)

- An open-source glue layer, which calls into the binary blob as well as the kernel API.

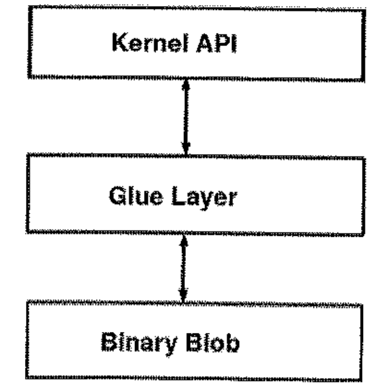


Figure 2.3: **Using binary blobs in drivers**

Well known examples of this method are employed by **Nvidia** graphics drivers, **ndiswrapper** wireless and NIC drivers, and various **ntfs** filesystem drivers.

With this method the driver writer can contain in the binary blob whatever code has been ported in from another operating system, or whatever code is wished to remain private; the first reason is most likely legal under **Linux** licensing, while the second is definitely not.

When such a driver is loaded the kernel becomes **tainted** which means it is impossible to debug properly because there is not source available for all the running code.

We strongly disapprove of this method for many reasons, but two are sufficient:

- Loading arbitrary binary code into the kernel is a recipe for disaster.
- Manufacturers promote this as **Linux** support and don't support the development of genuine open-source drivers.

Thus, we won't teach this method and we discourage even the use of such drivers, much less their development.

- Use of binary **firmware** is not the same as the methods described above. This firmware is data that could have been put in memory on the card, but vendors find it cheaper to have the operating system load it. Use of firmware does not cause tainting.
- The line between firmware and binary blobs is gray, and there are **Linux** distributions which have problems distributing drivers with require binary firmware, or which don't distribute the binary firmware itself.

2.4 How Applications Use Device Drivers

The **Unix** philosophy is to use a number of elementary methods connected through both piping and nesting to accomplish complex tasks.

User applications (and daemons) interact with peripheral devices using the same basic system calls irrespective of the specific nature of the device.

For the moment we'll leave networking device drivers out of the mix since they are not reached through filesystem entries; we'll concentrate on character drivers which have thinner layers between the applications and the device driver, and between the hardware and the device driver.

For each one of the limited number of these system calls, there is a corresponding **entry point** in the driver. The main ones for character drivers are:

`open()`, `release()`, `read()`, `write()`, `lseek()`, `ioctl()`, `mmap()`

Strictly speaking, the name of the system call and the entry point may differ, but in the above list the only one that does is the system call `close()` which becomes `release()` as an entry point. However, the return type of the system call as well as the arguments can be quite different than that of the entry point.

Note that there are other kinds of callback functions that may exist in a driver, which are **not** directly reached by user system calls:

- Loading and unloading the driver cause initializing and shutting down callbacks to get invoked.
- Higher layers of the kernel may call functions in the driver for such things as power management.
- The driver may execute a deferred task, such as after a given amount of time has elapsed, or a condition has become true.
- Interrupts may cause asynchronous execution of driver code, if the driver is registered to handle particular interrupts.

Once a device driver is loaded, therefore, its methods are all registered with the kernel, and it is event-driven; it responds to direct entries (which can be multiple and simultaneous) from user applications, and it executes code as requested by other kernel layers and in response to hardware provocation.

2.5 Walking Through a System Call

Let's see what actually happens when an application attempts to read from a device, which has already been opened. The application will open and then read with:

```
....
char *buf = malloc(nbytes);
fd = open ("/dev/mydriver", O_RDWR);
nread = read (fd, buf, nbytes);
....
```

(Of course we are giving only code fragments and being sloppy about error checking and all that!)

Let's concentrate on the `read()` call, which is simpler to trace than the `open()`. The `read()` is intercepted by **libc**, which knows how to make system calls, and the kernel is entered through the function `sys_read()`, which is in **/usr/src/linux/fs/read_write.c**:

```
2.6.31: 372 SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
2.6.31: 373 {
2.6.31: 374     struct file *file;
2.6.31: 375     ssize_t ret = -EBADF;
2.6.31: 376     int fput_needed;
2.6.31: 377
2.6.31: 378     file = fget_light(fd, &fput_needed);
2.6.31: 379     if (file) {
2.6.31: 380             loff_t pos = file_pos_read(file);
2.6.31: 381             ret = vfs_read(file, buf, count, &pos);
2.6.31: 382             file_pos_write(file, pos);
2.6.31: 383             fput_light(file, fput_needed);
2.6.31: 384     }
2.6.31: 385
2.6.31: 386     return ret;
2.6.31: 387 }
```

This function associates a `file` kernel data structure with `fd`, the user application file descriptor; the kernel works in terms of these structures, not file descriptors. It then passes off the work to `vfs_read()`:

```
2.6.31: 277 ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t
                *pos)
2.6.31: 278 {
2.6.31: 279     ssize_t ret;
2.6.31: 280
2.6.31: 281     if (!(file->f_mode & FMODE_READ))
2.6.31: 282             return -EBADF;
2.6.31: 283     if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
2.6.31: 284             return -EINVAL;
2.6.31: 285     if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
2.6.31: 286             return -EFAULT;
2.6.31: 287
2.6.31: 288     ret = rw_verify_area(READ, file, pos, count);
2.6.31: 289     if (ret >= 0) {
2.6.31: 290             count = ret;
2.6.31: 291             if (file->f_op->read)
2.6.31: 292                     ret = file->f_op->read(file, buf, count, pos);
2.6.31: 293             else
2.6.31: 294                     ret = do_sync_read(file, buf, count, pos);
2.6.31: 295             if (ret > 0) {
2.6.31: 296                     fsnotify_access(file->f_path.dentry);
2.6.31: 297                     add_rchar(current, ret);
2.6.31: 298             }
2.6.31: 299             inc_syscr(current);
2.6.31: 300     }
2.6.31: 301
2.6.31: 302     return ret;
2.6.31: 303 }
```

```
2.6.31: 304
2.6.31: 305 EXPORT_SYMBOL(vfs_read);
```

After checking to see if the file has been opened with permission for reading, and making other security checks, this function looks at the `file_operations` structure embedded in this structure (`file->f_op`) and if it has a read method defined (`file->f_op->read`), just passes the read request through to it. It then brings the return value of the read back to user-space.

Note that the functions `fget()`, `fput()` also increment and decrement a reference count of the file descriptor; the "light" version of these functions is faster for files already being used by a process.

None of the above is actually specific to device drivers; if we had opened a normal file, the method returned would have been that for the particular filesystem involved. Because our filesystem entry pointed to a device, we got device-specific methods instead.

Similar kinds of tracebacks can be performed for any of the entry points.

2.6 Error Numbers

Standard error numbers are defined in header files included from `linux/errno.h`. The first bunch come from **/usr/src/linux/include/asm-generic/errno-base.h**:

```
....
2.6.31:   4 #define EPERM        1  /* Operation not permitted */
2.6.31:   5 #define ENOENT       2  /* No such file or directory */
2.6.31:   6 #define ESRCH        3  /* No such process */
2.6.31:   7 #define EINTR        4  /* Interrupted system call */
2.6.31:   8 #define EIO          5  /* I/O error */
2.6.31:   9 #define ENXIO        6  /* No such device or address */
2.6.31:  10 #define E2BIG        7  /* Argument list too long */
2.6.31:  11 #define ENOEXEC      8  /* Exec format error */
2.6.31:  12 #define EBADF        9  /* Bad file number */
2.6.31:  13 #define ECHILD      10  /* No child processes */
2.6.31:  14 #define EAGAIN      11  /* Try again */
2.6.31:  15 #define ENOMEM      12  /* Out of memory */
2.6.31:  16 #define EACCES      13  /* Permission denied */
2.6.31:  17 #define EFAULT      14  /* Bad address */
....
```

Usually one returns `-ERROR`, while the error return for system calls is almost always `-1`, with the actual error code being stored in the global variable `errno` in user-space.

Thus, in user-space a typical code fragment would be:

```
#include <errno.h>

if ( ioctl(fd, CMD, arg) < 0 ){
     perror("MY_DRIVER_IOCTL_CALL");
     return (errno);
}
```


Remember, it is up to **you** to provide the proper error returns from your kernel entry points, which will cause `errno` to be set appropriately.

> Usually one can check the status of a function that has a pointer return value by simply checking for `NULL`. However, it doesn't make clear the cause of the error and the **Linux** kernel permits encoding the error code in the return value. The simple functions that deal with this situation are:
>
> ```
> void *ERR_PTR (long error);
> long IS_ERR (const void *ptr);
> long PTR_ERR (const void *ptr);
> ```
>
> where `ERR_PTR()` encodes the error, `IS_ERR()` checks for the presence of an error, and `PTR_ERR()` extracts the actual error code.

2.7 printk()

`printk()` is similar to the standard **C** function `printf()`, but has some important differences. A typical invocation might be:

```
printk(KERN_INFO "Your device driver was opened with Major Number = %d\n", major_number);
```

`printk()` has no floating point format support.

Every message in a `printk()` has a "loglevel" (if not explicitly given, a default level is applied). These levels are defined in **/usr/src/linux/include/linux/kernel.h**, and are:

```
#define KERN_EMERG    "<0>"  /* system is unusable               */
#define KERN_ALERT    "<1>"  /* action must be taken immediately */
#define KERN_CRIT     "<2>"  /* critical conditions              */
#define KERN_ERR      "<3>"  /* error conditions                 */
#define KERN_WARNING  "<4>"  /* warning conditions               */
#define KERN_NOTICE   "<5>"  /* normal but significant condition */
#define KERN_INFO     "<6>"  /* informational                    */
#define KERN_DEBUG    "<7>"  /* debug-level messages             */
```

The loglevel (or priority) forces an informational string to be pre-pended to your print statement.

The pseudo-file `/proc/sys/kernel/printk` lists the log levels set on the system: On most systems you'll get the numbers:

```
6       4       1       7
```

which have the following meanings respectively:

Table 2.3: **printk() logging levels**

Value	Meaning
console_loglevel	Messages with a higher priority than this will be printed to the console.
default_message_loglevel	Messages without an explicit priority will be printed with this priority.
minimum_console_level	Minimum (highest) value to which console_loglevel can be set.
default_console_loglevel	Default value for console_loglevel.

Note that a higher priority means a lower loglevel and processes can dynamically alter the console_loglevel; in particular a kernel fault raises it to 15.

Messages go into a circular log buffer, with a default length of 128 KB which can be adjusted during kernel compilation. The contents can be viewed with the **dmesg** utility; the file /var/log/dmesg contains the buffer's contents at system boot.

Where the messages go depends on whether or not **syslogd** and **klogd** are running and how they are configured. If they are not you can simply do cat /proc/kmsg. Generally they will go to /var/log/messages, but if you are running **X** you can't see them trivially. A good way to see them is to open a terminal window, and in that window type tailf /var/log/messages. You can also access the messages by clicking on System->Administration->System Log in your Desktop menus.

Ultimate control of kernel logging is in the hands of the daemons **syslogd** and **klogd**. These can control all aspects of the behaviour, including default levels and message destinations, directed by source and severity. There is actually a **man** page for **syslog** which also gives the **C**-language interface from within the kernel.

Note you can alter the various log levels through parameters to **syslogd**, but an even easier method is to exploit the /proc filesystem, by writing to it. The command

```
echo 8 > /proc/sys/kernel/printk
```

will cause all messages to appear on the console.

If the same line of output is repeatedly printed out, the logging programs are smart enough to compress the output, so if you do something like:

```
for (j=0; j<100; j++)
   printk(KERN_INFO "A message\n");
printk(KERN_INFO "another message\n");
```

what you will get out will be:


```
Dec 18 15:51:54 p3 kernel: A message
Dec 18 15:51:54 p3 last message repeated 99 times
Dec 18 15:51:54 p3 kernel: another message
```

Also you should note that you can only be assured that printk() will flush its output if the line ends with a "\n".

It is pretty easy to get overwhelmed with messages, and it is possible to limit the number of times a messages gets printed. The function for doing this is:

```
int printk_ratelimit (void);
```

and a typical use would be

```
if (printk_ratelimit())
    printk (KERN_WARNING "The device is failing\n");
```

Under normal circumstances you'll just get the normal printout. However, if the threshold is exceeded printk_ratelimit() will start returning zero and messages will fall on the floor.

The threshold can be controlled with modifying **/proc/sys/kernel/printk_ratelimit** and **/proc/sys/kernel/printk_ratelimit_burst**. The first parameter gives the minimum time (in jiffies) between messages, and the second the number of messages to send before rate-limiting kicks in.

An optional timestamp that can be printed with each line handled by printk() can be turned on by setting CONFIG_PRINTK_TIME in the configuration file. The time is printed out in seconds as in:

```
Jun 30 07:54:36 localhost kernel: [  707.921716] Hello: module loaded
Jun 30 07:54:36 localhost kernel: [  707.921765] Jiffies=1102136
Jun 30 07:54:43 localhost kernel: [  714.537416] Bye: module unloaded
Jun 30 07:54:43 localhost kernel: [  714.537422] Jiffies=1108757
```

2.8 Labs

Lab 1: Installing a Basic Character Driver

In this exercise we are going to compile, load (and unload) and test the a sample character driver (provided). In subsequent sections we will discuss each of these steps in detail.

Compiling: First you have to make sure you have the installed kernel source that you are going to use. The best way to compile kernel modules is to jump inside the kernel source to do it. This requires a simple Makefile, which need have only the line:

```
obj-m += lab1_chrdrv.o
```

If you then type

```
make -C<path to kernel source> M=$PWD modules
```

it will compile your module with all the same options and flags as the kernel modules in that source location.

Monitoring Output: If you are working at a virtual terminal or in non-graphical mode, you'll see the output of your module appear on your screen. Otherwise you'll have to keep an eye on the file `/var/log/messages`, to which the system logging daemons direct kernel print messages. The best way to do this is to open a terminal window, and in it type:

```
tailf /var/log/messages
```

Loading and Unloading: The easiest way to load and unload the module is with:

```
insmod lab1_chrdrv.ko
rmmod lab1_chrdrv
```

Try and load and unload the module. If you look at **/proc/devices** you should see your driver registered, and if you look at **/proc/modules** (or type **lsmod**) you should see that your module is loaded.

Creating a Device Node: Before you can actually use the module, you'll have to create the `device` node with which applications interact. You do this with the **mknod** command:

```
mknod /dev/mycdrv c 700 0
```

Note that the **name** of the device node (`/dev/mycdrv`) is irrelevant; the kernel identifies which driver should handle the system calls only by the **major number** (700 in this case.) The **minor number** (0 in this case) is generally used only within the driver.

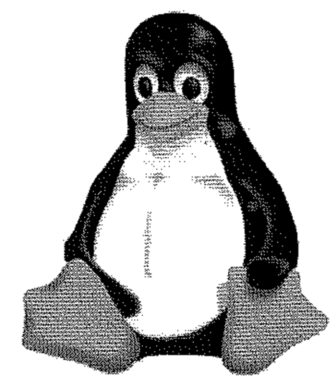
Using the Module: You should be able to test the module with simple commands like:

```
echo Some Input > /dev/mycdrv
cat somefile > /dev/mycdrv
dd if=/dev/zero of=/dev/mycdrv count=1
dd if=/dev/mycdrv count=1
```

We've only skimmed the surface; later we will consider the details of each of these steps.

Chapter 3

Modules I: Basics



We'll begin our discussion of modularization techniques under **Linux**. We'll define what a module is and describe the command level utilities used to manipulate them. We'll discuss how to compile, load, and unload modules, how to pass parameters to them, and how they interact with hotplugging.

3.1 What is a Module?

Modules are relocatable object code, that can be loaded or unloaded dynamically into or from the kernel as needed.

While most modules are device **drivers**, they need not be.

To a rough approximation, a module uses the **Linux** kernel like a shared library, linking in to it only through a list of symbols and functions which have been **exported** and thus made available to the module.

Modules may depend on each other and form a stack.

Type **lsmod** (or cat **/proc/modules**) to see what modules are presently loaded.

3.2 A Trivial Example - Hello World

Here is an example of a very trivial module. It does nothing but print a statement when it is loaded, and one when it is unloaded.

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

static int __init my_init (void)
{
    printk (KERN_INFO "Hello: module loaded at 0x%p\n", my_init);
    return 0;
}
static void __exit my_exit (void)
{
    printk (KERN_INFO "Bye: module unloaded from 0x%p\n", my_exit);
}

module_init (my_init);
module_exit (my_exit);

MODULE_AUTHOR ("A GENIUS");
MODULE_LICENSE ("GPL v2");
```

Almost all modules contain callback functions for initialization and cleanup, which are specified with the `module_init()` and `module_exit()` macros. These callbacks are automatically called when the module is loaded and unloaded. A module without a cleanup function cannot be unloaded.

In addition, use of these macros simplifies writing drivers (or other code) which can be used either as modules, or directly built into the kernel. Labeling functions with the attributes `__init` or `__exit` is a refinement to be discussed later.

Any module which does not contain an open source license (as specified with the `MODULE_LICENSE()` macro) will be marked as **tainted**: it will function normally but kernel developers will be hostile to helping with any debugging.

- You will still see modules with the outdated form:

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

int __init init_module (void)
{
    printk (KERN_INFO "Hello: init_module loaded at 0x%p\n", init_module);
    return 0;
}
void __exit cleanup_module (void)
{
    printk (KERN_INFO "Bye: cleanup_module loaded at 0x%p\n", cleanup_module);
}
```

- While direct use of the callback functions (`init_module()` and `cleanup_module()`) will still work, using them without employing the `module_init()` and `module_exit()` macros is deprecated.
- Many drivers in the kernel still use just the `init_module()`, `cleanup_module()` functions; it saves a few lines of code, especially for a driver that is always loaded as a module. While this is basically harmless, eventually use of this form will be extinguished.

3.3 Module Utilities

The following utilities run in user-space and are part of the **module-init-tools** package. They are not directly part of the kernel source. Each has a rather complete **man** page.

The configuration file **/etc/modprobe.conf** (as well as any files in the directory **/etc/modprobe.d**) is consulted frequently by the module utilities. Information such as paths, aliases, options to be passed to modules, commands to be processed whenever a model is loaded or unloaded, are specified therein. The possible commands are:

```
alias       wildcard modulename
options     modulename option ...
install     modulename command ...
remove      modulename command ...
include     filename
```

The `install` and `remove` commands can be used as substitutes for the default **insmod** and **rmmod** commands.

- All **Linux** distributions prescribe a methods for the automatic loading of particular modules on system startup. However, the use of **udev** in modern **Linux** distributions usually obviates such needs.
- On **Red Hat**-based systems the file **/etc/rc.modules** will be run (if it exists) out of **/etc/rc.d/rc.sysinit**. In this file any explicit module loading can be done through the full use of the module loading commands.
- On **Debian**-based systems any modules listed in **/etc/modules** will be loaded. (Only the names of the modules go in this file, not loading commands.) on **GENTOO** systems, the same role is played by the files in **/etc/modules.autoload.d**.
- On **SUSE**-based systems the file that needs to be modified is **/etc/sysconfig/kernel**.

lsmod

lsmod gives a listing of all loaded modules. The information given includes name, size, use count, and a list of referring modules. The content is the same as that in **/proc/modules**.

insmod

insmod links a loadable module into the running kernel, resolving all symbols. The `-f` option will try to force loading with a version mismatch between kernel and module.

insmod can be used to load parameters into modules. For example,

```
insmod my_net_driver.ko irq=10
```

rmmod

rmmod unloads modules from the running kernel. A list of modules may be given as in:

```
rmmod my_net_driver my_char_driver
```

Note no **.ko** extension is specified.

depmod

depmod creates a *Makefile*-like dependency file (**/lib/modules/KERNEL-VERSION-NUMBER /modules.dep**) based on the symbols contained in the modules explicitly mentioned on the command line, or in the default place.

depmod is vital to the use of **modprobe**. It is always run during boot. Under most circumstances it should be run as


```
depmod -ae
```

In order for **depmod** and **modprobe** to find modules they must be in prescribed places, under `/lib/modules`. The file **/etc/modprobe.conf** is consulted every time a module is loaded or when **depmod** is run.

When modules are built in external directories and installed with the `modules_install` target, they are placed in the `extra` subdirectory.

modprobe

modprobe can load (or unload with the `-r` option) a stack of modules that depend on each other, and can be used instead of **insmod**.

It can also be used to try a list of modules, and quit whenever one is first found and successfully loaded. It is also heavily dependent on **/etc/modprobe.conf**.

Whenever there are new modules added, or there is a change in location `depmod` should be run.

The **modtuils** package requires use of the `.ko` extension with **insmod**, but **modprobe** and **rmmod** require no extension.

Parameters can be passed to **modprobe** in the same way they are passed to **insmod**; i.e., you can do something like

```
modprobe my_net_driver irq=10
```

3.4 Passing Parameters

Parameters to be passed to modules must be explicitly marked as such and type checking is done. For example,

```
int irq = 12;
module_param (irq, int, 0);
```

There are a number of macros which can be used:

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

module_param (name, type, perm);
module_param_named (name, value, type, perm);
module_param_array (name, type, num, perm);
module_param_string (name, string, len, perm);
```

In the basic `module_param()` macro, `name` is the variable name, `type` can be `byte, short, ushort, int, uint, long, ulong, charp, bool, invbool`.

The `perm` parameter is a permissions mask which is used for an accompanying entry in the **sysfs** filesystem. If you are not interested in **sysfs**, a value of `0` will suffice. Typically one can use the value

S_IRUGO (0444) for read permission for all users. (See **/usr/src/linux/include/linux/stat.h** for all possibilities.) If a write permission is given, the parameter may be altered by writing to the `sysfs` filesystem entry associated with the module, but note that the module will not be notified in any way when the value changes! In this case the permission might be `S_IRUGO | S_IWUSR` (0644).

The `module_param_named()` variation has `name` as the string used to describe the variable when loading, which does not have to be the same as `value` (which is **not** the actual value of the parameter but rather the name of the parameter as used in the module.) Note that these are macros, and neither argument has to appear within quotes.

In the `module_param_array()` macro the integer variable `num` gives the number elements in the array which can be used as in the following example:

```
insmod my_module irq=3,4,5
```

The `module_param_string()` macro is for passing a string directly into a `char` array.

With this method it is possible to build your own data types; it is extensible. For more details see **http://lwn.net/Articles/22197/**.

A feature of this method is that it still works when a driver, or other kernel facility, is compiled as built-in rather than as a module. Kernels earlier than 2.6 required use of a separate set of functions (using the `__setup()` macros) to pass parameters to the kernel on the boot command line. This makes it even easier to write code that can be used as either a module or built-in without changing the source.

The way to pass such a parameter to the kernel as a boot parameter is to prefix its name with the name of the module and a .; thus the kernel boot command line might look like:

```
linux root=LABEL=/ ... my_module.irq=3,4,5 ...
```

A list of all known passable parameters can be found at **/usr/src/linux/Documentation/kernel-parameters.txt**.

There are a number of related macros, defined in **/usr/src/linux/include/linux/module.h** that can be used in modules:

```
MODULE_AUTHOR(name);
MODULE_DESCRIPTION(desc);
MODULE_SUPPORTED_DEVICE(name);
MODULE_PARM_DESC(var,desc);
MODULE_FIRMWARE(filename);
MODULE_LICENSE(license);
MODULE_VERSION(version);
```

The information stored thereby is generated by running the command **modinfo**.

3.5 Compiling a Module

In order to compile modules you must have the kernel source installed; or at least those parts of it which are required. Those should always be found under **/lib/modules/$(uname -r)/build**.

The simple-minded way to compile a module would require specifying the right flags and options, and pointing to the correct kernel headers. However, this method has long been deprecated and in the 2.6 kernel series it has become impossible to compile completely outside the kernel source tree.

Compilation of modules for the 2.6 kernel **requires** a kernel source which has either been through a compilation stage, or at least has been through a `make prepare`, as this is required to generate necessary configuration and dependency information. One also needs, of course a `.config` containing the kernel configuration.

The approved approach is still to work outside the kernel source tree, but to jump into it to do the compilation. For this you'll need at least a minimal `Makefile` with at least the following in it:

```
obj-m += trivial.o
```

If you then type

```
make -C/lib/modules/$(uname -r)/build M=$PWD modules
```

it will compile your module with all the same options and flags as the kernel modules in that source location (for the currently running kernel). To compile for a kernel other than the one that is running, you just need to place the proper argument with the `-C` option.

Installing the modules in the proper place so they can be automatically found by utilities such as **depmod**, requires the `modules_install` target:

```
make -C/lib/modules/$(uname -r)/build M=$PWD modules_install
```

By default the output is brief; to make it more verbose you can set the environmental variables `V=1` or `KBUILD_VERBOSE=1`. You could do this, for example, by typing `export KBUILD_VERBOSE=1; make` or `make V=1 ...`, etc.

If it is necessary to split the source into more than one file, then the `-r` option to **ld** (which is automatically invoked by **gcc**) can be used. A simple example (for the Makefile) would be:

```
obj-m += mods.o
mods-objs := mod1.o mod2.o
```

In the main directory of the solutions, you will find a script titled **genmake** which can automatically generate proper makefiles; it can be a great time-saver! Here is what it looks like:

```
#!/bin/bash

# Automatic kernel makefile generation
# Jerry Cooperstein, coop@coopj.com, 2/2003 - 1/2009
# License: GPLv2

OBJS=""   # list of kernel modules (.o files)
K_S=""    # list of kernel modules (.c files)
U_S=""    # list of userland programs (.c files)
U_X=""    # list of userland programs (executables)
```

```
T_S=""    # list of userland programs (.c files) that use pthreads
T_X=""    # list of userland programs (executables) that use pthreads
ALL=""    # list of all targets

CFLAGS_U_X="-O2 -Wall -pedantic"    # compile flags for user programs
CFLAGS_T_X=$CFLAGS_U_X" -pthread"   # compile flags for threaded user programs

# set the default kernel source to the running one; otherwise take from
# first command line argument

if [ "$KROOT" == "" ] ; then
    KROOT=/lib/modules/$(uname -r)/build
    [ ! -d "$KROOT" ] && KROOT=/usr/src/linux-$(uname -r)
fi

# abort if the source is not present

KMF=$KROOT/Makefile
KERNELSRC=$(grep "^KERNELSRC" $KMF | awk ' {print $3;}')

if [ "$KERNELSRC" != "" ] ; then
    echo Primary Makefile is not in $KROOT, using $KERNELSRC
    KMF=$KERNELSRC/Makefile
fi

if [ ! -d "$KROOT" ] || [ ! -f "$KMF" ] ; then
    echo kernel source directory $KROOT does not exist or has no Makefile
    exit 1
fi

# additional flags?

if [ "$KINCS"        != "" ] ; then
        CFLAGS_U_X="$CFLAGS_U_X -I$KROOT/include"
        CFLAGS_T_X="$CFLAGS_T_X -I$KROOT/include"
fi

# extract the VERSION info from the Makefile

KV=$(grep "^VERSION =" $KMF | awk ' {print $3;}')
KP=$(grep "^PATCHLEVEL =" $KMF | awk ' {print $3;}')
KS=$(grep "^SUBLEVEL =" $KMF | awk ' {print $3;}')
KE=$(grep "^EXTRAVERSION =" $KMF | awk ' {print $3;}')
KERNEL=$KV.$KP.$KS$KE

echo KERNEL=$KERNEL, KV=$KV KP=$KP KS=$KS KE=$KE

# check on the major release version

if [ $KP != 6 ] ; then
    echo KROOT=$KROOT is not 2.6, exiting
    exit 1
fi

# construct lists of kernel and user sources and targets
```



```
# skip empty directories
if [ "$(find . -maxdepth 1 -name "*.c")" == "" ] ; then
    echo No need to make Makefile: no source code
    exit
fi

for names in *.c ; do

# exclude files with NOMAKE or .mod.c files

 if [ "$(grep NOMAKE $names)" ] || [ "$(grep vermagic $names)" ] ; then
    echo "$names is being skipped, it is not a module or program"
    else
    if [ "$(grep \<linux\/module.h\> $names )" ] ; then
        FILENAME_DOTO=$(basename $names .c).o
        OBJS=$OBJS" $FILENAME_DOTO"
        K_S=$K_S" $names"
    else
# is it  a pthread'ed program?
        if [ "$(grep '\<pthread.h\>' $names)" ] ; then
            FILENAME_EXE=$(basename $names .c)
            T_X=$T_X" $FILENAME_EXE"
            T_S=$T_S" $names"
        else
            U_X=$U_X" $(basename $names .c)"
            U_S=$U_S" $names"
        fi
    fi
 fi
done

CLEANSTUFF="$U_X $T_X"

# maybe there are no kernel modules

if [ "$OBJS" != "" ] ; then
    CLEANSTUFF="$CLEANSTUFF"" Module.symvers modules.order"
fi

# get ALL the targets

if [ "$U_X" != "" ] ; then ALL=$ALL" userprogs"; fi
if [ "$T_X" != "" ] ; then ALL=$ALL" threadprogs"; fi
if [ "$OBJS" != "" ] ; then ALL=$ALL"  modules" ; fi

# echo if you are curious :>

echo K_S=$K_S OBJS=$OBJS U_S=$U_S U_X=$U_X T_S=$T_S T_X=$T_X

#######################################################################################
# We're done preparing, lets build the makefile finally!

# get rid of the old Makefile, build a new one
```

```
rm -f Makefile

echo "### Automatic Makefile generation by 'genmake' script       ####" >>Makefile
echo "### Copyright, Jerry Cooperstein, coop@coopj.com 2/2003 - 1/2009 ####" >>Makefile
echo "### License: GPLv2 ###"                                              >>Makefile

if [ "$K_S" != "" ] ; then
    echo -e "\nobj-m += $OBJS" >> Makefile
    echo -e "\nexport KROOT=$KROOT" >> Makefile
    if [ "$ARCH" != "" ] ; then
        echo -e "\nexport ARCH=$ARCH" >> Makefile
    fi
fi
echo -e "\nallofit: $ALL" >> Makefile

if [ "$K_S" != "" ] ; then
    echo "modules:" >> Makefile
    echo -e "\t@\$(MAKE) -C \$(KROOT) M=\$(PWD) modules" >> Makefile
    echo "modules_install:" >> Makefile
    echo -e "\t@\$(MAKE) -C \$(KROOT) M=\$(PWD) modules_install" >> Makefile
    echo "kernel_clean:" >> Makefile
    echo -e "\t@\$(MAKE) -C \$(KROOT) M=\$(PWD) clean" >> Makefile
fi

if [ "$U_X" != "" ] ; then
    echo -e "\nuserprogs:" >> Makefile
    echo -e "\t@\$(MAKE) \\" >> Makefile
    echo -e "\t\tCFLAGS=\"$CFLAGS_U_X\" \\" >> Makefile
    if [ "$LDLIBS" != "" ] ; then
        echo -e "\t\tLDLIBS=\"$LDLIBS\" \\" >> Makefile
    fi
    if [ "$CPPFLAGS" != '' ] ; then
    echo -e "\t\tCPPFLAGS=\"$CPPFLAGS\" \\" >> Makefile
    fi
    echo -e "\t$U_X" >> Makefile
fi

if [ "$T_X" != "" ] ; then
    echo -e "\nthreadprogs:" >> Makefile
    echo -e "\t@\$(MAKE) \\" >> Makefile
    echo -e "\t\tCFLAGS=\"$CFLAGS_T_X\" \\" >> Makefile
    if [ "$LDLIBS" != "" ] ; then
        echo -e "\t\tLDLIBS=\"$LDLIBS\" \\" >> Makefile
    fi
    if [ "$CPPFLAGS" != '' ] ; then
        echo -e "\t\tCPPFLAGS=\"$CPPFLAGS\" \\" >> Makefile
    fi
    echo -e "\t$T_X" >> Makefile
fi

if [ "$K_S" != "" ] ; then
    echo -e "\nclean: kernel_clean" >> Makefile
else
    echo -e "\nclean:" >> Makefile
fi
```



```
echo -e "\trm -rf $CLEANSTUFF" >> Makefile

exit

#####################################################################################
```

3.6 Modules and Hot Plug

When the system is aware that a new device has been added or is present at boot, it is also often furnished with information describing the device. This usually includes (but is not limited to) a unique **vendor id** and **product id**.

Drivers can specify which devices they can handle, and when modules are installed on the system, catalogues are updated. Thus, the **hot plug** facility (in user-space) is able to consult these tables and automatically load the required device driver, if it is not already present.

For this to work the driver has to use the macro:

```
MODULE_DEVICE_TABLE(type,name)
```

where `type` indicates the type of driver and `name` points to an array of structures, each entry of which specifies a device. The exact structure depends on the type of device. For example:

```
static const struct pci_device_id skge_id_table[] = {
 { PCI_DEVICE(PCI_VENDOR_ID_3COM, PCI_DEVICE_ID_3COM_3C940) },
 { PCI_DEVICE(PCI_VENDOR_ID_3COM, PCI_DEVICE_ID_3COM_3C940B) },
....
 { PCI_VENDOR_ID_LINKSYS, 0x1032, PCI_ANY_ID, 0x0015, },
 { 0 }
};
MODULE_DEVICE_TABLE(pci, skge_id_table);
```

Note the use of the `PCI_DEVICE()` macro; each type of subsystem has such macros to aid in filling out the table of structures.

The allowed types and the structures for them are delineated in **/usr/src/linux/include/linux/mod_devicetable.h** and include:

```
usb
pci
ieee1394
pcmcia
i2c
input
eisa
pnp
serio
```

and each one has a structure associated with it, such as `struct usb_device_id`, `struct ieee1394_device_id`, etc.

Furthermore, when one runs the make `modules_install` step one updates the appropriate files in **/lib/modules/$(uname -r)**. such as `modules.pcimap`, `modules.usbmap`, etc.

When a new device is added to the system, these files are consulted to see if it is a known device, and if so and the required driver is not already loaded, **modprobe** is run on the proper driver.

3.7 Labs

Lab 1: Module parameters

Write a module that can take an integer parameter when it is loaded with **insmod**. It should have a default value when none is specified.

Load it and unload it. While the module is loaded, look at its directory in **/sys/module**, and see if you can change the value of the parameter you established.

Lab 2: Initialization and cleanup functions.

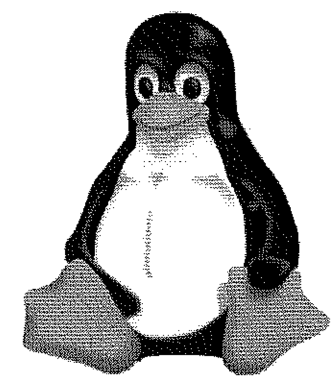
Take any simple module, and see what happens if instead of having both initialization and cleanup functions, it has:

- Only an initialization function.
- Only a cleanup function.
- Neither an initialization nor a cleanup function.

In these cases see what happens when you try to load the module, and if that succeeds, when you try to unload it.

Chapter 4

Character Devices



We'll begin our detailed discussion of building character device drivers. We'll talk about device nodes, how to create them, access them, and register them with the kernel. We'll discuss the **udev/HAL** interface. Then we'll describe in detail the important **file_operations** data structure and itemize in detail the driver entry points it points to. Two other important data structures, the **file** and **inode** structures are also considered. Finally we show how modules keep track of their usage count.

4.1 Device Nodes

Character and block devices have filesystem entries associated with them. These **nodes** can be used by user-level programs to communicate with the device, whose driver can manage more than one device node.

Examples of device nodes:

```
lrwxrwxrwx  1 root root         3 May 25 01:56 cdwriter -> hda
brw-rw----  1 coop floppy   2,  0 May 25 01:56 fd0
brw-rw----  1 coop floppy   2,  4 May 25 01:56 fd0D360
brw-rw----  1 coop floppy   2, 16 May 25 01:56 fd0D720
crw-rw----  1 root lp      99,  0 May 25 01:56 parport0
crw-rw----  1 root lp      99,  1 May 25 01:56 parport1
crw-rw----  1 root lp      99,  2 May 25 01:56 parport2
```

Device nodes are made with:

```
mknod [-m mode] /dev/name <type> <major> <minor>

e.g., mknod -m 666 /dev/mycdrv c 254 1
```

or from the `mknod()` system call.

4.2 Major and Minor Numbers

The **major** and **minor** numbers identify the driver associated with the device. Generally speaking, all device nodes of the same type (block or character) with the same major number use the same driver.

The **minor** number is used **only** by the device driver to differentiate between the different devices it may control. These may either be different instances of the same kind of device, (such as the first and second sound card, or hard disk partition) or different modes of operation of a given device (such as different density floppy drive media.)

The major and minor numbers are stored together in a variable of type `dev_t`, which has 32 bits, with 12 bits reserved for the major number, and 20 bits for the minor number.

The internal bit layout is complicated for historical reasons, and one is not guaranteed that it will not change in future kernel versions. Thus one should always use the following macros to construct (or deconstruct) major and minor numbers from a `dev_t` structure:

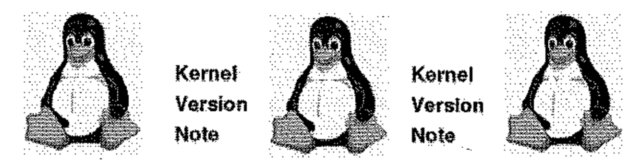
Table 4.1: **Device node macros**

Macro	Meaning
`MAJOR(dev_t dev);`	Extract the major number.
`MINOR(dev_t dev);`	Extract the minor number.
`MKDEV(int major, int minor);`	Return a `dev_t` built from major and minor numbers

One can also use the inline convenience functions:

```
unsigned iminor(struct inode *inode); /* = MINOR(inode->i_rdev) */
unsigned imajor(struct inode *inode); /* = MAJOR(inode->i_rdev) */
```

when one needs to work with `inode` structures.



- In the 2.4 kernel, device numbers were packed in the `kdev_t` type, which was limited to 16 effective bits, even divided between minor and major numbers, so that each was limited to the range `0 - 255`.
- In the 2.4 kernel once a driver registered a major number, no other driver could be registered with the same major number, and all minor numbers belonged to the driver.
- In the 2.6 kernel, however, one registers a **range** of minor numbers which can be less than all available, and indeed two concurrently loaded drivers can have the same major number, as long as they have distinct minor number ranges.

A list of the major and minor numbers pre-associated with devices can be found in **/usr/src/linux /Documentation/devices.txt**. (Note the major numbers 42, 120-127 and 240-254 are reserved for local and experimental use.) Symbolic names for assigned major numbers can be found in **/usr/src/linux/include/linux/major.h**. Requesting further device number reservations is probably prohibited, as more modern methods use **dynamical allocation**.

Note that device numbers have meaning in user-space as well; in fact some **Posix** system calls such as `mknod()` and `stat()` have arguments with the `dev_t` data type, or utilize structures that do. For example:

```
$ stat vmlinuz-2.6.26-rc5

  File: 'vmlinuz-2.6.26-rc5'
  Size: 2849808         Blocks: 5592       IO Block: 1024 regular file
Device: 805h/2053d      Inode: 22090       Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (   0/ root)
Access: 2008-06-07 13:20:51.000000000 -0500
Modify: 2008-06-07 13:20:51.000000000 -0500
Change: 2008-06-07 13:20:51.000000000 -0500
```

shows the file resides on the disk partition with major number 8 and minor number 5 (`/dev/sda5`), which is listed at `805h` (hexadecimal) or `2053d` (decimal).

4.3 Reserving Major/Minor Numbers

Adding a new driver to the system (i.e., registering it) means assigning a major number to it, usually during the device's initialization routine. For a character driver one calls:

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

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

where `first` is the first device number being requested, of a range of `count` contiguous numbers; usually the minor number part of `first` would be 0, but that is not required.

`name` is the device name, as it will appear when examining **/proc/devices**. Note it is **not** the same as the node name in `/dev` that your driver will use. (The kernel decides which driver to invoke based on the major/minor number combination, not the name.)

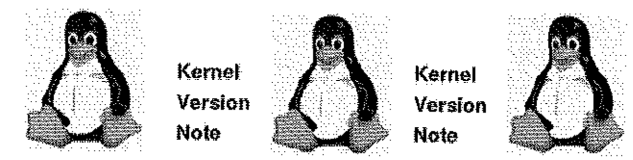
A return value of 0 indicates success; negative values indicate failure and the requested region of device numbers will not be available. Note that **mknod** will still have to be run to create the appropriate device node(s).

It is important when undoing the registration to remove the association with device numbers, once they are no longer needed. This is most often done in the device cleanup function with:

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

void unregister_chrdev_region (dev_t first, unsigned int count);
```

Note that this will **not** remove the node itself.

If you fail to unregister a device, you'll get a segmentation fault the next time you do `cat /proc/devices`. It is pretty hard (although not impossible) to recover from this kind of error without a system reboot.



- In the 2.4 kernel only 8-bit major and minor numbers were available, and the functions for registering and de-registering were:

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

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

- Dynamic allocation was accomplished by specifying 0 as a a major number; the return value gave the supplied major number which was obtained by decreasing from 254 until an unused number was found. (When dynamic allocation was not requested, the return value upon success was 0, which was confusing.)
- We'll discuss the `struct file_operations` pointer argument shortly, which delineates the methods used by the driver.
- Only one device driver could use a given major number at at a time; in the 2.6 kernel it is required that only major/minor number set is unique.
- This interface, while more limited that the new one, is very widely used in the kernel and there is no great rush to eliminate it. However, new drivers should use the improved 32-bit methods.

Dynamic Allocation of Major Numbers

Choosing a unique major number may be difficult: dynamic allocation of the device numbers is the proper method for all new drivers and can be used to avoid collisions. This is accomplished with the function:

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

int alloc_chrdev_region (dev_t *first, unsigned int firstminor, unsigned int count,
                         char *name);
```

where `first` is now passed by address as it will be filled in by the function. The new argument, `firstminor` is obviously the first requested minor number, (usually 0.) The de-registration of the device numbers is the same with this method.

The disadvantage of dynamic allocation is that the proper node can not be made until the driver is loaded. Furthermore, one usually needs to remove the node upon unloading of the driver module.

Thus some scripting is required around both the module loading and unloading steps.

While it would be possible to have a module do an `exec()` call to `mknod` and jump out to user space, this is never done; kernel developers feel strongly that making nodes belongs in user-land, not the kernel.

Even better, you can use the **udev** facility to create a node from within your module. We'll show you how to do this later.

4.4 Accessing the Device Node

Under **Unix**-like operating systems, such as **Linux**, applications access peripheral devices using the same functions as they would for ordinary files. This is an essential part of the ***everything is a file*** philosophy. For example, listening to a sound would involve **reading** from the device node associated with the sound card (generally `/dev/audio`).

There are a limited number of **entry points** into device drivers, and in most cases there is a one to one mapping of the **system calls** applications make and the entry point in the driver which is exercised when the call is made.

For a given class of devices, such as **character** or **block**, the entry points are the same irrespective of the actual device itself. In the case of character drivers, the mapping is relatively direct; in the case of block drivers there is more indirection; i.e., several layers of the kernel may intercede between the system call and the entry point; a read would involve the virtual filesystem, the actual filesystem, and cache layers before requests to get blocks of data on or off a device are made to the driver through a `read()` or `write()` system call.

The following are the main operations that can be performed on character device nodes by programs in user-space:

```
int     open  (const char *pathname, int flags);
int     close (int fd);
ssize_t read  (int fd, void *buf, size_t count);
ssize_t write (int fd, const void *buf, size_t count);
int     ioctl (int fd, int request, ...);
off_t   lseek (int fd, off_t offset, int whence);
void    *mmap (void *start, size_t length, int prot, int flags, int fd, off_t offset );
int poll (struct pollfd *fds, nfds_t nfds, int timeout);
```

These entry points all have **man** pages associated with them.

The device driver has entry points corresponding to these functions; however names and arguments may differ. In the above list, for example, the system call `close()` will lead to the entry point `release()`.

Remember that applications can exert these system calls indirectly; for instance by using the standard I/O library functions, `fopen()`, `fclose()`, `fread()`, `fwrite()`, and `fseek()`.

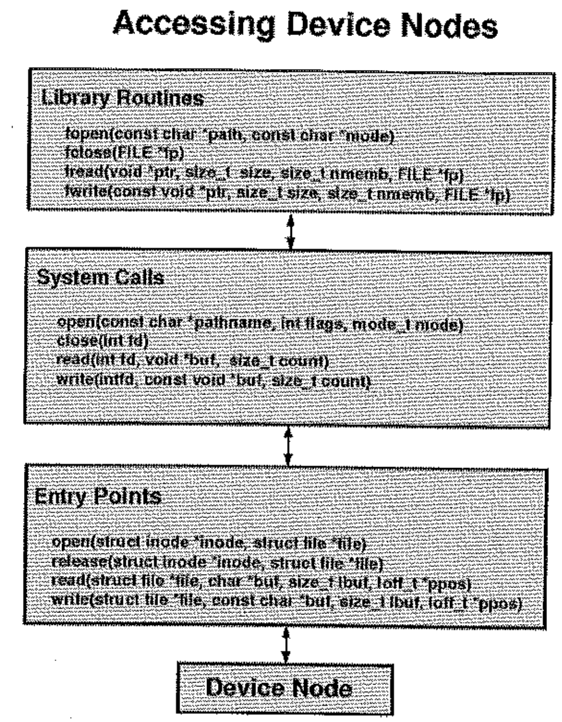


Figure 4.1: **Accessing device nodes**

4.5 Registering the Device

So far all we have done is reserve a range of device numbers for the exclusive use of our driver. More work has to be done before the device can be used.

Character devices are associated with a `cdev` structure, as defined in **/usr/src/linux/include /linux/cdev.h:**

```
struct cdev {
        struct kobject kobj;
        struct module *owner;
        struct file_operations *ops;
        struct list_head list;
        dev_t dev;
        unsigned int count;
};
```

Normally you won't work directly with the internals of this structure, but reach it through various utility functions. In particular we'll see how the `owner` and `ops` pointers are used.

A number of related functions which are needed to work with character devices are:

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

```
struct cdev *cdev_alloc (void);
void cdev_init (struct cdev *p, struct file_operations *fops);
void cdev_put (struct cdev *p);
struct kobject *cdev_get (struct cdev *p);
int cdev_add (struct cdev *p, dev_t first, unsigned count);
void cdev_del (struct cdev *p);
```

These structures should always be allocated dynamically, and then initialized with code like:

```
struct cdev *mycdev = cdev_alloc ();
cdev_init (mycdev, &fops);
```

This code allocates memory for the structure, initializes it, and sets the owner and ops fields to point to the current module, and the proper `file_operations` table.

The driver will go **live** when one calls:

```
cdev_add (mycdev, first, count);
```

This function should not be called until the driver is ready to handle anything that comes its way. The inverse function is

```
cdev_del (mycdev)
```

and after this is called the device is removed from the system and the `cdev` structure should never be accessed after this point.

4.6 udev and HAL

The methods of managing device nodes became clumsy and difficult as **Linux** evolved. The number of device nodes lying in **/dev** and its subdirectories reached numbers in the 15,000 - 20,000 range in most installations during the 2.4 kernel series. Nodes for all kinds of devices which would never be used on most installations were still created by default, as distributors could never be sure exactly which hardware would be needed.

Of course many developers and system administrators trimmed the list to what was actually needed, especially in embedded configurations, but this was essentially a manual and potentially error-prone task.

Note that while device nodes are not normal files and don't take up significant space on the filesystem, having huge directories slowed down access to device nodes, especially upon first usage. Furthermore, exhaustion of available major and minor numbers required a more modern and dynamic approach to the creation and maintenance of device nodes.

Ideally, one would like to register devices by name, However, major and minor numbers can not be gotten rid of altogether, as **Posix** requires them.

The **udev** method creates device nodes on the fly as they are needed. There is no need to maintain a ton of device nodes that will never be used. The **u** in **udev** stands for **user**, and indicates that most of the work of creating, removing, and modifying devices nodes is done in user-space.

udev handles the dynamical generation of device nodes but it does not handle the discovery or management of them. This requires the **Hardware Abstraction Layer**, or **HAL**, which is a project of **freedesktop.org** (**http://www.freedesktop.org/wiki/Software/hal**).

HAL uses the **D-BUS** (device bus) infrastructure, as provided by the **HAL** daemon (**haldaemon**). It maintains a dynamic database of all connected hardware devices and is closely coupled to the hotplug facility. The command `lshal` will dump out all the information that **HAL** currently has in its database. There are a number of configuration files on the system (in **/usr/share/hal** and **/etc/hal**) which control behaviour and set exceptions.

The cleanest way to use **udev** is to have a pure system; the `/dev` directory is empty upon the initial kernel boot, and then is populated with device nodes as they are needed. When used this way, one must boot using an **initrd** or **initramfs** image, which may contain a set of preliminary device nodes as well as the **udev** program itself.

As devices are added or removed from the system, working with the hotplug subsystem, **udev** acts upon notification of events to create and remove device nodes. The information necessary to create them with the right names, major and minor numbers, permissions, etc, are gathered by examination of information already registered in the **sysfs** pseudo-filesystem (mounted at `/sys`) and a set of configuration files.

The main configuration file is **/etc/udev/udev.conf**. It contains information such as where to place device nodes, default permissions and ownership etc. By default rules for device naming are located in the **etc/udev/rules.d** directory. By reading the **man** page for **udev** one can get a lot of specific information about how to set up rules for common situations.

Creation and removal of a device node dynamically, from within the driver using **udev**, is done by the use of the following functions defined in **/usr/src/linux/include/linux/device.h**:

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

struct class  *class_create  (struct module *owner, const char *name);
struct device *device_create (struct class *cls, struct device *parent, dev_t devt,
                              const char *fmt, ...);
void device_destroy (struct class *cls, dev_t dev);
void class_destroy (struct class *cls);
```

Generally, the `parent` is `NULL` which means the class is created at the top level of the hierarchy. A code fragment serves to show the use of these functions:

```
static struct class *foo_class;

/* create node in the init function */
   foo_class = class_create (THIS_MODULE, "my_class");
   device_create (foo_class, NULL, first, "%s%d", "mycdrv", 1);

/* remove node in the exit function */
   device_destroy (foo_class,first);
   class_destroy (foo_class);
```

One has to be careful to do whatever is necessary to make the device usable before the device node is created, to avoid race conditions.

4.7 file_operations Structure

The `file_operations` structure is defined in **/usr/src/linux/include/linux/fs.h**, and looks like:

```
2.6.31:1486 struct file_operations {
2.6.31:1487     struct module *owner;
2.6.31:1488     loff_t (*llseek) (struct file *, loff_t, int);
2.6.31:1489     ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
2.6.31:1490     ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
2.6.31:1491     ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long,
                loff_t);
2.6.31:1492     ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long,
                loff_t);
2.6.31:1493     int (*readdir) (struct file *, void *, filldir_t);
2.6.31:1494     unsigned int (*poll) (struct file *, struct poll_table_struct *);
2.6.31:1495     int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
2.6.31:1496     long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
2.6.31:1497     long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
2.6.31:1498     int (*mmap) (struct file *, struct vm_area_struct *);
2.6.31:1499     int (*open) (struct inode *, struct file *);
2.6.31:1500     int (*flush) (struct file *, fl_owner_t id);
2.6.31:1501     int (*release) (struct inode *, struct file *);
2.6.31:1502     int (*fsync) (struct file *, struct dentry *, int datasync);
2.6.31:1503     int (*aio_fsync) (struct kiocb *, int datasync);
2.6.31:1504     int (*fasync) (int, struct file *, int);
2.6.31:1505     int (*lock) (struct file *, int, struct file_lock *);
2.6.31:1506     ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *,
                int);
2.6.31:1507     unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned
                long, unsigned long, unsigned long);
2.6.31:1508     int (*check_flags)(int);
2.6.31:1509     int (*flock) (struct file *, int, struct file_lock *);
2.6.31:1510     ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
                size_t, unsigned int);
2.6.31:1511     ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
                size_t, unsigned int);
2.6.31:1512     int (*setlease)(struct file *, long, struct file_lock **);
2.6.31:1513 };
2.6.31:1514
2.6.31:1515 struct inode_operations {
2.6.31:1516     int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
2.6.31:1517     struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata
                *);
2.6.31:1518     int (*link) (struct dentry *,struct inode *,struct dentry *);
2.6.31:1519     int (*unlink) (struct inode *,struct dentry *);
2.6.31:1520     int (*symlink) (struct inode *,struct dentry *,const char *);
2.6.31:1521     int (*mkdir) (struct inode *,struct dentry *,int);
2.6.31:1522     int (*rmdir) (struct inode *,struct dentry *);
2.6.31:1523     int (*mknod) (struct inode *,struct dentry *,int,dev_t);
2.6.31:1524     int (*rename) (struct inode *, struct dentry *,
2.6.31:1525                     struct inode *, struct dentry *);
2.6.31:1526     int (*readlink) (struct dentry *, char __user *,int);
2.6.31:1527     void * (*follow_link) (struct dentry *, struct nameidata *);
2.6.31:1528     void (*put_link) (struct dentry *, struct nameidata *, void *);
```



```
2.6.31:1529     void (*truncate) (struct inode *);
2.6.31:1530     int (*permission) (struct inode *, int);
2.6.31:1531     int (*setattr) (struct dentry *, struct iattr *);
2.6.31:1532     int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
2.6.31:1533     int (*setxattr) (struct dentry *, const char *,const void *,size_t,int);
2.6.31:1534     ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
2.6.31:1535     ssize_t (*listxattr) (struct dentry *, char *, size_t);
2.6.31:1536     int (*removexattr) (struct dentry *, const char *);
2.6.31:1537     void (*truncate_range)(struct inode *, loff_t, loff_t);
2.6.31:1538     long (*fallocate)(struct inode *inode, int mode, loff_t offset,
2.6.31:1539                       loff_t len);
2.6.31:1540     int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start,
2.6.31:1541                   u64 len);
2.6.31:1542 };
```

and is a **jump table** of **driver entry points**, with the exception of the first field, `owner`, which is used for module reference counting.

This structure is used for purposes other than character drivers, such as with filesystems, and so some of the entries won't be used in this arena. The same is true with the `file` and `inode` structures to be discussed shortly.

The file operations structure is initialized with code like:

```
struct file_operations fops = {
        .owner   = THIS_MODULE,
        .open    = my_open,
        .release = my_close,
        .read    = my_read,
        .write   = my_write,
        .ioctl   = my_ioctl,
};
```

According to the C99 language standard, the order in which fields are initialized is irrelevant, and any unspecified elements are `NULL`-ed.

These operations are associated with the device with the `cdev_init()` function, which places a pointer to the `file_operations` structure in the proper `cdev` structure. Whenever a corresponding system call is made on a device node owned by the device, the work is passed through to the driver; e.g., a call to open on the device node causes the `my_open()` method to be called in the above example.

- If no method is supplied in the `file_operations` structure, there are two possibilities for what will occur if the method is invoked through a system call:
  - The method will fail: An example is `mmap()`.
  - A generic default method will be invoked: An example is `llseek()`. Sometimes this means the method will always succeed: Examples are `open()` and `release()`.

4.8 Driver Entry Points

`struct module *owner;`

The only field in the structure that is not a method. Points to the module that owns the structure and is used in reference counting and avoiding race conditions such as removing the module while the driver is being used. Usually set to the value `THIS_MODULE`.

`loff_t (*llseek) (struct file *filp, loff_t offset, int whence);`

changes the current read/write position in a file, returning the new position. Note that `loff_t` is 64-bit even on 32-bit architectures.

If one wants to inhibit seeking on the device (as on a pipe), one can unset the `FMODE_LSEEK` bit in file `file` structure (probably during the `open()` method) as in:

```
file->f_mode = file->f_mode & ~FMODE_LSEEK;
```

`ssize_t (*read) (struct file *filp, char __user *buff, size_t size, loff_t *offset);`

reads data from the device, returning the number of bytes read. An error is a negative value; zero may mean end of device and is not an error. You may also choose to block if data is not yet ready and the process hasn't set the non-blocking flag.

A simple `read()` entry point might look like:

```
static ssize_t mycdrv_read (struct file *file, char __user *buf, size_t lbuf, loff_t * ppos)
{
    int nbytes = lbuf - copy_to_user (buf, kbuf + *ppos, lbuf);
    *ppos += nbytes;
    printk (KERN_INFO "\n READING function, nbytes=%d, pos=%d\n", nbytes, (int) *ppos);
    return nbytes;
}
```

In this simple case a read merely copies from a buffer in kernel-space (`kbuf`) to a buffer in user-space (`buf`.) But one can not use `memcpy()` to perform this because it is improper to de-reference user pointers in kernel-space; the address referred to may not point to a valid page of memory at the current time, either because it hasn't been allocated yet or it has been swapped out.

Instead one must use the `copy_to_user()` and `copy_from_user()` functions (depending on direction) which take care of these problems. (We'll see later there are more advanced techniques for avoiding the extra copy, including memory mapping, raw I/O, etc.)

The kernel buffer will probably be dynamically allocated, since the in-kernel per-task stack is very limited. This might be done with:

```
#include <linux/slab.h>
char *kbuf = kmalloc (kbuf_size, GFP_KERNEL);
....
kfree (kbuf);
```

where the limit is 1024 pages (4 MB on **x86**).

If using the `GFP_KERNEL` flag, memory allocation may block until resources are available; if `GFP_ATOMIC` is used the request is non blocking. We will discuss memory allocation in detail later.

The position in the device is updated by modifying the value of `*ppos` which points to the current value. The return value is the number of bytes successfully read; this is a case where a positive return value is still success.

`ssize_t (*write) (struct file *filp, const char __user *buff, size_t size, loff_t *offset);`

writes data to the device, returning the number of bytes written. An error is a negative value; zero may mean end of device and is not an error. You may also choose to block if the device is not yet ready and the process hasn't set the non-blocking flag.

The same considerations apply about not directly using user-space pointers. Here one should use

```
int nbytes = lbuf - copy_from_user (kbuf + *ppos, buf, lbuf);
```

for the same reason

`int (*readdir) (struct file *filp, void *, filldir_t filldir);`

should be `NULL` for device nodes; used only for directories, and is used by filesystem drivers, which use the same `file_operations` structure.

`unsigned int (*poll) (struct file *filp, struct poll_table_struct *ptab);`

checks to see if a device is readable, writable, or in some special state. In user-space this is accessed with both the `poll()` and `select()` calls. Returns a bit mask describing device status.

`int (*ioctl) (struct inode *inode, struct file *filp, unsigned int, unsigned long);`

is the interface for issuing device-specific commands. Note that some `ioctl` commands are not device-specific and are intercepted by the kernel without referencing your entry point.

`long (*unlocked_ioctl) (struct file *filp, unsigned int, unsigned long);`

Unlike the normal `ioctl()` entry point, the big kernel lock is not taken before and released after calling. New code should use this entry point; if present the old one will be ignored.

`int (*mmap) (struct file *filp, struct vm_area_struct *vm);`

requests a mapping of device memory to a process's memory. If you don't implement this method, the system call will return `-ENODEV`.

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

opens a device. If set to `NULL` opening the device always succeeds, but the driver is not notified. The `open()` method should:

- Check for hardware problems like the device not being ready.
- Initialize the device if it is the first time it is being opened.
- If required, note the minor number.
- Set up any data structure being used in `private_data` field of the `file` data structure.

`int (*flush) (struct file *filp);`

is used when a driver closes its copy of a file descriptor for a device. It executes and waits for any outstanding operations on a device. Rarely used. Using `NULL` is safe.

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

closes the node. Note when a process terminates all open file descriptors are closed, even under abnormal exit, so this entry may be called implicitly. The `release()` method should reverse the operations of `open()`:

- Free any resources allocated by open.
- If it is the last usage of the device, take any shutdown steps that might be necessary.

`int (*fsync) (struct file *filp, struct dentry *dentry, int datasync);`

is used to flush any pending data.

`int (*fasync) (int, struct file *filp, int);`

checks the devices `FASYNC` flag, for *asynchronous* notification. Use `NULL` unless your driver supports asynchronous notification.

`int (*lock) (struct file *filp, int, struct file_lock *lock);`

is used to implement file locking; generally not used by device drivers, only files.

`ssize_t (*aio_read) (struct kiocb *iocb, const struct iovec *iov, unsigned long niov, loff_t pos);`

`ssize_t (*aio_write)(struct kiocb *iocb, const struct iovec *iov, unsigned long niov, loff_t pos);`

`int (*aio_fsync) (struct kiocb *, int datasync);`

These implement **asynchronous** methods for I/O. If not supplied, the kernel will always use the corresponding synchronous methods.

`ssize_t (*sendfile) (struct file *filp, loff_t *offset, size_t, read_actor_t, void *);`

Implements copying from one file descriptor to another without separate read and write operations, minimizing copying and the number of system calls made. Used only when copying a file through a socket. Unused by device drivers.

`ssize_t (*sendpage) (struct file *filp, struct page *, int, size_t, loff_t *offset, int);`

Inverse of `sendfile()`; used to send data (a page at a time) to a file. Unused by device drivers.

`unsigned long (*get_unmapped_area) (struct file *filp, unsigned long, unsigned long, unsigned long, unsigned long);`

Find an address region in the process's address space that can be used to map in a memory segment from the device. Not normally used in device drivers.

`int (*check_flags)(int);`

A method for parsing the flags sent to a driver through `fcntl()`.

`int (*dir_notify) (struct file *filp, unsigned long arg);`

Invoked when `fcntl()` is called to request directory change notifications. Not used in device drivers.

`int (*flock) (struct file *, int, struct file_lock *);`

Used for file locking; Not used in device drivers.

4.9 The file and inode Structures

The `file` and `inode` data structures are defined in **/usr/src/linux/include/linux/fs.h**. Both are important in controlling both device nodes and normal files.

The `file` structure has nothing to do with the `FILE` data structure used in the standard **C** library; it never appears in user-space programs.

A new `file` structure is created whenever the `open()` call is invoked on the device, and gets passed to all functions that use the device node. This means there can be multiple `file` structures associated with a device simultaneously, as most devices permit multiple opens. The structure is released and the memory associated with it is freed during the `release()` call.

Some important structure members:

Table 4.5: **file structure elements**

Field	Meaning
`struct path f_path`	Gives information about the file directory entry, including a pointer to the inode.
`const struct file_operations f_op`	Operations associated with the file. Can be changed when the method is invoked again.
`f_mode_t f_mode`	Identified by the bits `FMODE_READ` and `FMODE_WRITE`. Note the kernel checks permissions before invoking the driver.
`loff_t f_pos`	Current position in the file; a 64-bit value. While the final argument to the `read()` and `write()` entry points usually points to this, the `llseek()` entry should update `f_pos`, but the `read()` and `write()` entry points should update the argument. (Use of `f_pos` for this purpose is incorrect because the `pread()`, `pwrite()` system calls use the same read and write methods but do not have this linkage.)
`unsigned int f_flags`	`O_RDONLY`, `O_NONBLOCK`, `O_SYNC`, etc. Needs to be checked for non-blocking operations.

void * private_data	Can be used to point to allocated data. Can be used to preserve state information across system calls. The pointer to this is set to NULL before the open() call, so your driver can use this to point to whatever it wants, such as an allocated data structure. In this case you must remember to free the memory upon release(). Note there will be a unique instance of this structure for each time your device is opened.

Note that a pointer to the file_operations structures lies inside the structure. To obtain a pointer to the inode structure you have to descend through the f_path element which is a structure of type:

```
struct path {
    struct vfsmount *mnt;
    struct dentry *dentry;
};
```

with the inode field contained in the dentry structure. So you'll often see references like:

```
struct file *f;
f->f_path.dentry->d_inode
```

or using a backwards compatible macro as:

```
f->f_dentry->d_inode
```

but this macro is slated for removal.

Unlike the file structure, there will only be one inode structure pointing to a given device node; each open descriptor (and corresponding internal file structure representation) on the device node will in turn to point to that same inode structure.

While the inode structure contains all sorts of information about the file it points to, here it happens to be a device node, and very few of the fields are of interest for character drivers. Two of importance are:

Table 4.6: **inode structure elements**

Field	Meaning
dev_t i_rdev	Contains the actual device number from with the major and minor numbers can be extracted.
dev_t i_cdev	Points back to the basic character driver structure.

4.10 Module Usage Count

The kernel needs to keep track of how many times a module is being referenced by user-space processes. (This is unrelated to how many times the module is being used by other modules.) It is impossible to remove a module with a non-zero reference count.

Once upon a time modules were expected to do most of the bookkeeping on their own, incrementing the usage count whenever a module was used by a process, and decrementing it when it was done. This procedure was difficult to accomplish without incurring errors and race conditions.

As an improvement, module usage is now kept track of by higher levels of the kernel rather than manually. For this to work one needs to set the owner field in the appropriate data structure for the type of module being considered. For instance for a character device driver or a filesystem driver, this would be the file_operations structure. One can set this through:

```
static const struct file_operations fops = {
    .owner= THIS_MODULE,
    .open=  my_open,
    ...
}
```

Now the kernel will take care of the bookkeeping automatically.

Other examples of such structures containing tables of callback functions, or entry points, with owner fields include block_device_operations and fb_ops.

If there is a need to manually modify a module's usage count (to prevent unloading while the module is being used) one can use the functions:

```
int try_module_get (struct module *module);
void module_put (struct module *module);
```

Note that a call like try_module_get(THIS_MODULE) can fail if the module is in the process of being unloaded, in which case it returns 0; otherwise it returns 1. These functions are defined to have no effect when module unloading is not allowed as a kernel option during configuration.

The reference count itself is embedded in the module data structure and can be obtained with the function

```
unsigned int module_refcount (struct module *mod);
```

and would usually be invoked as something like:

```
printk(KERN_INFO "Reference count= %d\n", module_refcount(THIS_MODULE));
```

4.11 Labs

Lab 1: Improving the Basic Character Driver

Starting from **sample_driver.c**, extend it to:

- Keep track of the number of times it has been opened since loading, and print out the counter every time the device is opened.
- Print the major and minor numbers when the device is opened.

To exercise your driver, write a program to read (and/or write) from the node, using the standard Unix I/O functions (`open()`, `read()`, `write()`, `close()`).

After loading the module with **insmod** use this program to access the node.

Track usage of the module by using **lsmod** (which is equivalent to typing `cat /proc/modules`.)

Lab 2: Private Data for Each Open

Modify the previous driver so that each opening of the device allocates its own data area, which is freed upon release. Thus data will not be persistent across multiple opens.

Lab 3: Seeking and the End of the Device.

Adapt one of the previous drivers to have the read and write entries watch out for going off the end of the device.

Implement a `lseek()` entry point. See the **man** page for `lseek()` to see how return values and error codes should be specified.

For an extra exercise, unset the `FMODE_LSEEK` bit to make any attempt to seek result in an error.

Lab 4: Dynamical Node Creation (I)

Adapt one of the previous drivers to allocate the device major number dynamically.

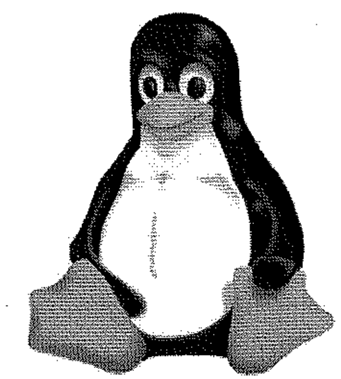
Write loading and unloading scripts that ascertain the major number assigned and make and remove the node as required.

Lab 5: Dynamical Node Creation (II)

Adapt the previous dynamic registration driver to use **udev** to create the device node on the fly.

Chapter 5

Kernel Configuration and Compilation



We'll examine the layout of the **Linux** kernel source. We'll consider methods of browsing the source. We'll also give the procedures for configuring, compiling, and installing updated or modified kernels. Finally we'll consider the use of **initrd** images.

5.1 Installation and Layout of the Kernel Source

The source for the **Linux** kernel must be made easily available by all distributors. Both the newest and older kernel versions are generally available for download. (Remember that `finger @www.kernel.org` will give a quick enumeration of the most recent kernel versions.)

The pristine source for all kernel versions can always be obtained from directly from the kernel maintainers at **http://www.kernel.org**, or from the distributors, most of whom make (possibly quite extensive) changes to the source. These changes must also be freely available.

The exact location of the source is on your system is neither mandated nor important. When external modules have to be built against the source, the directory `/lib/modules/$(uname -r)/build` either contains the actual source, or is a symbolic link pointing to it.

For convenience, we'll often pretend the kernel source resides at `/usr/src/linux`, and for the purpose of displaying code, create a symbolic link from there to the real code. This should not be construed as a recommendation to do this on normal development systems. Under `/usr/src/linux` (or the real location) we find:

Table 5.1: **Layout of the kernel source**

Directory	Purpose
`arch`	`i386, ia64, alpha, arm, sparc, sparc64, mips, mips64, m68k, ppc, s390...` Architecture specific code for boot, synchronization. memory and process management.
`kernel`	Generic main kernel routines.
`mm`	Generic memory management code. swapping, mmaping, kernel malloc, etc.
`init`	Generic kernel start-up code.
`drivers`	`char, block, net, scsi, fs, cdrom, pci ...` Device drivers sorted by type.
`sound`	**ALSA** (**A**dvanced **L**inux **S**ound **A**rchitecture), including sound card drivers.
`block`	Low-level infrastructure for the block device layer. Specific block device drivers are under `drivers/block`.
`fs`	Filesystems, with subdirectories for each type.
`net`	Ethernet, ip, decnet, ipx, ipv4, ipv6, appletalk and other network code.
`security`	Security models, including **SELinux**.
`crypto`	Cryptographic algorithms.
`lib`	Some standard library routines, mostly for strings.
`ipc`	System V Inter-Process Communications code.
`usr`	User-space interaction; so far only **intramfs** code.
`scripts`	Various scripts used to compile and package kernels.
`Documentation`	Various documentation resources; sometimes not up to date.
`virt`	Virtualization infrastructure.
`firmware`	Firmware that is packaged with kernel.
`samples`	Sample kernel code used for tracing, profiling and debugging purposes.
`tools`	User-space tools, used for performance counting.
`include`	System header files.

Here is a count of lines in the source code for the most recent vanilla kernel. (For each directory all subdirectories are included.)

Directory	.S files	.h files	.c files	TOTAL
.	310430	1992776	8461916	10765122
./drivers	1776	801738	5014569	5818083
./arch	305894	654445	1192799	2153138
./fs	0	72900	821466	894366
./net	0	15967	560563	576530
./sound	218	38800	484524	523542
./include	0	385398	0	385398
./kernel	0	2773	140979	143752
./mm	0	266	62889	63155
./security	0	4721	41171	45892
./crypto	0	9814	33396	43210
./lib	0	817	31656	32473
./scripts	0	2838	22263	25101
./block	0	167	16405	16572
./tools	0	1565	13772	15337
./Documentation	0	0	9155	9155
./ipc	0	177	7320	7497
./virt	0	164	4103	4267
./init	0	76	2994	3070
./firmware	2425	0	268	2693
./samples	0	150	1032	1182
./usr	117	0	592	709

Most of the lines of code are for drivers, either for peripherals or different types of filesystems. A fair comparison with other operating systems should observer that the sources for the **X**-window system and the various Desktops, etc., are not included. However, we have included all architectures.

Here is how the total number of lines has changed with recent kernel versions:

Kernel Version	TOTAL LINES
2.6.18	7082952
2.6.19	7308043
2.6.20	7400843
2.6.21	7522286
2.6.22	7744727
2.6.23	7818168
2.6.24	8082358
2.6.25	8395801
2.6.26	8535933
2.6.27	8690888
2.6.28	9128690
2.6.29	9871260
2.6.30	10419567
2.6.31	10765122

5.2 Kernel Browsers

One often has to browse the kernel source in order to understand the inner workings of the kernel, compare kernel versions etc. Often the best tools for doing so are the simple text utilities such as **grep** and **find**.

One modern tool is the **Linux Cross Reference Browser** (**lxr**) which can be accessed at **http://lxr.linux.no**. This website contains browseable source code repositories for virtually every linux kernel ever produced, as well as source code and instructions for a local installation of **lxr**.

The master **lxr** repository uses version 0.9.4 of the browser, which while very robust is relatively slow and more difficult to install compared to older versions. A sample installation of the simpler version 0.3.1 can be found at **http://users.sosdg.org/ qiyong/lxr/source**

Local use of **lxr** requires running a web server (typically **apache**) and several hundred MB of disk space per kernel being indexed.

A purely text-based browser is offered by the **cscope** utility, which is a standard offering on most **Linux** systems. To use on the kernel sources one need merely run `make cscope` in the kernel source directory, creating the various index files needed, and then simply run **cscope** in the main kernel source directory. From then on the use is interactive and intuitive.

Another approach is afforded by the use of **GNU Global**, which can be obtained from **http://www.gnu.org/software/global**. Many distributions offer this as a package. Once **global** is installed one need go only to the kernel source directory and do **gtags** `; htags`, wait until the cross-indexing is done and then navigate the results using your favorite browser, simply by pointing to `/usr/src/linux/HTML/index.html`. One disadvantage of **global** is its use of over 2 GB of disk space per kernel.

All these methods will work, and there are some others such as just doing **make tags** and using the generic tags files that can be used by **emacs** and **vi** experts. For what it's worth, we confess to having a preference for **lxr** (older versions) because of the easy comparison of different kernel versions.

5.3 Kernel Configuration Files

Kernels provided by **Linux** distributors usually differ from those whose sources are directly obtained from the official "vanilla" kernel repository at **http://www.kernel.org**. Patches, sometimes quite extensive, have been made to the kernel source, including the addition of new features and device drivers that have not yet made it into the "official" kernel tree, as well as bug fixes and security enhancements.

The default configuration for the vanilla sources has only a few kinds of hardware turned on (such as one network card) as well as various subsystems turned off, with the actual default choices probably reflecting the actual hardware Linus Torvalds has (or had at one point), such as his choice of sound card, network driver etc.

When you configure the kernel you produce a configuration file, `.config`, in the main kernel source directory. If configured to do so, the contents of the `.config` file can be stored right inside the kernel. If the `CONFIG_IKCONFIG_PROC` option is turned on, information can be read out directly from `/proc/config.gz`.

Running the `scripts/extract-ikconfig` utility on a kernel image (**compressed** or **uncompressed**) built with `CONFIG_IKCONFIG` turned on, causes a dump of the configuration file. (Beware, this utility needs to be run from the main kernel source directory, or requires some minor modifications to work.)

If you don't have the full kernel source install, you can still find your `.config` file in the directory `/lib/modules/$(uname -r)/build/`, as well as find an additional copy in the `/boot` directory.

In these configurations, drivers for almost all conceivable hardware are compiled as kernel modules. This is the correct thing to do because one cannot know in advance precisely what hardware the end user will have, so all possibilities must be prepared for.

On the other hand, by configuring only the hardware actually present the kernel compilation can be sped up considerably. In addition, the configuration process goes much faster as you only have to turn on what you need.

5.4 Rolling Your Own Kernel

For purposes of experimentation you may want to try using another kernel, in particular you may want to compile and install a vanilla kernel from **http://www.kernel.org**. In order to to do this you need two files:

- The compressed kernel source (e.g., `linux-2.6.31.tar.bz2`)
- A configuration file (e.g., `config-2.6.31_x86_64`)

The first of these files can be obtained from **http://www.kernel.org**, and the second can be obtained from **http://www.coopj.com/LDD**

The following script (`DO_KERNEL.sh`), included in your solutions, can take care of all necessary steps of unpacking, configuring, compiling, and installing. To do everything just type

```
$ DO_KERNEL.sh 2.6.31 linux-2.6.31.tar.bz2 config-2.6.31_x86_64
```

If the kernel source has already been unpacked and configured, just go to the kernel source directory and running the script with no arguments will take care of compilation and installation.

```
#!/bin/bash

# Script to compile and install the Linux kernel and modules.
# written by Jerry Cooperstein (2002-2009)
# Copyright GPL blah blah blah

function get_SOURCE(){
    KERNEL=$1
    TARFILE=$2
    CONFIGFILE=$3
    LKV=linux-$KERNEL
    KCONFIG=$LKV/.config
    [ ! -f "$TARFILE" ] && echo no  "$TARFILE, aborting" && exit
    [ ! -f "$CONFIGFILE" ] && echo no "$CONFIGFILE, aborting" && exit
```

```
    echo -e "\nbuilding: Linux $KERNEL kernel from $TARFILE
               Using: $CONFIGFILE as the configuration file\n"
    set -x
# Decompress kernel, assumed in bz2 form
    tar jxf $TARFILE
# Copy over the .config file
    cp $CONFIGFILE $KCONFIG
# Change to the main source directory
    cd $LKV
    set +x
}

# determine which distribution we are on
function get_SYSTEM(){
    [ "$SYSTEM" != "" ] && return
    SYSTEM=
    [ "$(grep -i Red\ Hat /proc/version)" != "" ] && SYSTEM=REDHAT && return
    [ "$(grep -i Ubuntu   /proc/version)" != "" ] && SYSTEM=UBUNTU && return
    [ "$(grep -i debian   /proc/version)" != "" ] && SYSTEM=DEBIAN && return
    [ "$(grep -i suse     /proc/version)" != "" ] && SYSTEM=SUSE   && return
    [ "$(grep -i gentoo   /proc/version)" != "" ] && SYSTEM=GENTOO && return
}

# find out what kernel version this is
function get_KERNELVERSION(){
    for FIELD in VERSION PATCHLEVEL SUBLEVEL EXTRAVERSION ; do
        eval $(sed -ne "/^$FIELD/s/ //gp" Makefile)
    done
# is there a local version file?
    [ -f ./localversion-tip ] && \
        EXTRAVERSION="$EXTRAVERSION$(cat localversion-tip)"
    KERNEL=$VERSION.$PATCHLEVEL.$SUBLEVEL$EXTRAVERSION
}

# determine where to place vmlinuz, System.map, initrd image, config file
function get_BOOT(){
    if [ "$BOOT" == "" ] ; then
        PLAT=$(uname -i)
        BOOT=/boot
        [ "$PLAT" == "i386" ]   && [ -d /boot/32 ] && BOOT=/boot/32
        [ "$PLAT" == "x86_64" ] && [ -d /boot/64 ] && BOOT=/boot/64
    fi
}

# parallelize, speed up for multiple CPU's
function get_MAKE(){
    NCPUS=$(grep ^processor /proc/cpuinfo | wc -l)
    MAKE="make -j $(( $NCPUS * 2))"
}

# if it is a NVIDIA'able  kernel, compile nvidia.ko
function  dealwith_NVIDIA(){
     [ "$(lsmod | grep nvidia)" ] && \
     [ "$(which --skip-alias dealwith_nvidia)" ] && \
         dealwith_nvidia $KERNEL
```



```
}

function makeinitrd_REDHAT(){
# Construct mkinitrd image
# RHEL5.3 2.6.18 kernels mucked this up
    if [ "$(grep CONFIG_MD_RAID45 ./.config)" != "" ] \
        || [ "$(grep without-dmraid /sbin/mkinitrd)" == "" ] ; then
        mkinitrd -v -f              $BOOT/initrd-$KERNEL.img $KERNEL
    else
        mkinitrd -v -f --without-dmraid   $BOOT/initrd-$KERNEL.img $KERNEL
    fi
# Update /boot/grub/grub.conf
    cp /boot/grub/grub.conf /boot/grub/grub.conf.BACKUP
    grubby --copy-default  \
        --remove-kernel=$BOOT/vmlinuz-$KERNEL \
        --add-kernel=$BOOT/vmlinuz-$KERNEL \
        --initrd=$BOOT/initrd-$KERNEL.img \
        --title=$BOOT/$KERNEL
# if it is a NVIDIA'able  kernel, compile nvidia.ko
    dealwith_NVIDIA
}

function makeinitrd_DEBIAN(){
    make install
    if [ -f $BOOT/initrd.img-$KERNEL ] ; then
        update-initramfs -u -k $KERNEL
    else
        update-initramfs -c -k $KERNEL
    fi
    update-grub
}

function makeinitrd_UBUNTU(){
    makeinitrd_DEBIAN
}

function makeinitrd_SUSE(){
    make install
}

function makeinitrd_GENTOO(){
    make install
    genkernel          ramdisk --kerneldir=$PWD
}
function makeinitrd_(){
    echo System $SYSTEM is not something I understand, can not finish
    exit
}

#############################################################################
# Start of the work

NARGS="$#"
[ "$NARGS" == "3" ]  && get_SOURCE $1 $2 $3
```

```
get_KERNELVERSION
get_BOOT
get_SYSTEM
get_MAKE

echo building: Linux $KERNEL kernel,  and placing in: $BOOT on a $SYSTEM system

# set shell to abort on any failure and echo commands
set -e -x

# Do the main compilation work, kernel and modules
$MAKE

# Install the modules
$MAKE modules_install

# Install the compressed kernel, System.map file, config file,
cp arch/x86/boot/bzImage    $BOOT/vmlinuz-$KERNEL
cp System.map               $BOOT/System.map-$KERNEL
cp .config                  $BOOT/config-$KERNEL

# making initrd and updating grub is very distribution dependendent:

echo I am building the initrd image and modifying grub config on $SYSTEM

makeinitrd_"$SYSTEM"
```

We've been sloppy because one should really separate the steps of **compiling** and **installing**. Any user should be able to compile the kernel, while only the superuser should be able to install the kernel and modules. Compiling as superuser exposes one to potential bugs which can screw up the system; historically this has indeed happened on development kernels.

While the compiling step is distribution-independent the installation steps can be quite different in two of the steps; construction of the **initrd** or **initramfs** image and its naming; and editing the **grub** (or **lilo**) configuration file. The above script may not work properly in every conceivable circumstance, but it should be more than adequate in common installations.

5.5 initrd and initramfs

On many if not most **Linux** systems certain kernel modules need to be loaded before the root filesystem can be fully mounted. In most circumstances the modules required are those needed to mount the proper block devices on which the filesystem resides. This has always been true for **SCSI** systems, but has come to include **journalling** filesystems, such as **ext3**.

In the early days of **Linux**, if the necessary block drivers and filesystems were built-in to the kernel it was unnecessary to go through the two-phase boot procedure we are about to discuss. However, if your system is configured to use the **udev** facility to generate device nodes automatically (which most modern **Linux** distributions are), the two step procedure can be less painful than avoiding it and is customary in all major distributions.

On the other hand, for embedded devices it is rarely necessary to use **udev**, and a one stage boot process is usually the most efficient procedure.

In kernel versions before 2.6, the **initrd** (initial ram disk) method was used to solve this problem. The initial ram disk (usually built with a utility with a name like **mkinitrd**) contained the necessary driver modules, a version of **insmod** to load them, a copy of **udev** if necessary, and a simple shell (such as **nash** on **Red Hat**-based systems) used to execute any other necessary commands.

In the 2.6 kernel, a newer method called **initramfs** (initial ram filesystem) is the default. It is leaner and differs in many technical details. In particular, an initramfs image can be embedded in the kernel itself. It can also be supplied on the kernel command line with the `initrd=` specification. For this method to work the kernel has to be built with ram disk and initrd support.

The **man** pages for **initrd** and **mkinitrd** contain sufficient documentation to explain the details. (Even though the method is now **initramfs** the name of the utility has not been changed.) Here we simply note you'll need a line in your **lilo** or **grub** configuration file pointing to the initial ram disk, and to prepare a new ram disk for a new kernel. For example, on **Red Hat**-based systems you need only to do something like:

```
mkinitrd initrd-2.6.31.img 2.6.31
```

This figures out exactly which files, utilities and modules you need for the first phase. The image will be in the form of a **cpio** archive of a root filesystem which can be unpacked and examined with:

```
$ mkdir temp && cd temp
$ gunzip -c ../initrd-2.6.31.img | cpio -idv
$ ls -lR
```

which gives:

```
total 32
drwx------ 2 coop coop 4096 Jun 29 08:17 bin
drwxrwxr-x 3 coop coop 4096 Jun 29 08:17 dev
drwx------ 2 coop coop 4096 Jun 29 08:17 etc
-rwx------ 1 coop coop 1528 Jun 29 08:17 init
drwx------ 3 coop coop 4096 Jun 29 08:17 lib
drwx------ 2 coop coop 4096 Jun 29 08:17 proc
lrwxrwxrwx 1 coop coop    3 Jun 29 08:17 sbin -> bin
drwx------ 2 coop coop 4096 Jun 29 08:17 sys
drwx------ 2 coop coop 4096 Jun 29 08:17 sysroot

./bin:
total 2852
-rwx------ 1 coop coop  525400 Jun 29 08:17 insmod
lrwxrwxrwx 1 coop coop      10 Jun 29 08:17 modprobe -> /sbin/nash
-rwx------ 1 coop coop 2390592 Jun 29 08:17 nash

./dev:
total 4
drwx------ 2 coop coop 4096 Jun 29 08:17 mapper
lrwxrwxrwx 1 coop coop    4 Jun 29 08:17 ram -> ram1

./dev/mapper:
total 0
```

```
./etc:
total 0

./lib:
total 184
-rw------- 1 coop coop 56288 Jun 29 08:17 ahci.ko
-rw------- 1 coop coop 45944 Jun 29 08:17 ehci-hcd.ko
drwx------ 2 coop coop  4096 Jun 29 08:17 firmware
-rw------- 1 coop coop 33336 Jun 29 08:17 ohci-hcd.ko
-rw------- 1 coop coop  7992 Jun 29 08:17 pata_marvell.ko
-rw------- 1 coop coop 32088 Jun 29 08:17 uhci-hcd.ko

./lib/firmware:
total 0

./proc:
total 0

./sys:
total 0

./sysroot:
total 0
```

A script called **init** (in the main directory on the image filesystem) contains the detailed startup and load procedures.

If you wish to modify the filesystem produced by **mkinitrd** (perhaps you are not happy with the order in which things get loaded), then you can do so, and re-compress the image when you are finished. You can recreate a new image with:

```
find . | cpio --quiet -c -o  > ../imagenew && gzip ../imagenew
```

and then renaming the compressed image appropriately.

- The above discussion is somewhat **Red Hat**-centric. For example, **debian**-based systems (including **Ubuntu**) use a utility called **update-initramfs** to both produce an image and update the **grub** configuration. **Gentoo** systems use **genkernel ramdisk ..** for the same purpose.
- Some distributions construct only a minimal initial images, others throw in the kitchen sink. There has been discussion of moving much of this work into the kernel source tree, and indeed most of the infrastructure already exists. However, distributors have interests, histories, and concerns about this which may make a coherent cross-distributor approach be far off.

5.6 Labs

Lab 1: Building a Kernel

In this exercise you will build a **Linux** kernel, tailored to specific needs of hardware/software. You won't actually modify any of the source for the **Linux** kernel; however, you will select features and decide which modules are built.

Use whatever exact file names and version numbers are appropriate for the sources you have, rather than what is specified below.

Step 1: Obtain and install the source

Depending on your **Linux** distribution you may already have the source installed for your currently running kernel. You should be able to do this by looking at the **/lib/modules/kernel-version/** directory and seeing if it has active links to **build** or **source** directories. If not you'll have to obtain the kernel source in the method detailed by your distribution.

If you are using a vanilla source, then download it from **http://www.kernel.org** and then unpack it with:

```
$ tar jxvf linux-2.6.31.tar.bz2
```

(putting in the proper file and kernel version of course.)

Step 2: Make sure other ingredients are up to date.

The file **/usr/src/linux/Documentation/Changes** highlights what versions of various system utilities and libraries are needed to work with the current source.

Step 3: Configuring the Kernel

You can use any of the following methods:

- **make config**

  A purely text-based configuration routine.
- **make menuconfig**

  An **ncurses** semi-graphical configuration routine.
- **make xconfig**

  An **X**-based fully-graphical configuration routine, based on the **qt** graphical libraries.
- **make gconfig**

  Also an **X**-based fully-graphical configuration routine, based on the **GTK** graphical libraries, which has a somewhat different look..

You'll probably want to use **make xconfig** or **make gconfig**, as these have the nicest interfaces. At any rate, the content and abilities of all the methods are identical. They all produce a file named **.config**, which contains your choices. (It is generally advised not to edit this file directly unless you really know what you are doing!)

If you have an old configuration, you can speed up the process by doing:

```
$ make oldconfig
```

which takes your old configuration and asks you only about new choices. If you want to get the default choices as they come out of **kernel.org**, you can obtain the initial configuration with

```
$ make defconfig
```

Also note that if you are going to a new version through applying a patch, you can use the **patch-kernel** script by going to the source directory and doing:

```
$ scripts/patch-kernel . < patch directory >
```

- The **ketchup** utility, obtainable from **http://www.selenic.com/ketchup**, is very useful for going from one kernel version to another.
- **ketchup** will even download patches and/or full sources as they are needed, and can check source integrity.
- For instance upgrading from 2.6.24 to 2.6.31 would involve going to the source directory and just typing:

  ```
  $ ketchup -G 2.6.31
  ```

Take your time configuring the kernel. Read the help items to learn more about the possibilities available. Several choices you should make (for this class) are:

- Under **Processor type and features:**

  Pick the proper CPU (Choosing too advanced a processor make cause a boot failure.)
- Under **Loadable module support:**

  Turn **on** "Enable loadable module support."

  Turn **on** "Module unloading."
- Under **Block Devices:**

  Turn **on** "Loopback device support."

  Turn **on** "RAM disk support"

  Turn **on** "initial RAM disk (initrd) support"

- Under **Multi-device Support (RAID and LVM)**:

  Turn **on** "Device Mapper Support"
- Under **Instrumentation Support**:

  Turn **on** "Profiling Support" and "Oprofile"

  Turn **on** "Kprobes"
- Under **Kernel Hacking:**

  Turn **on** "Magic SysRq key".

  Turn **on** "Debug Filesystem".

Make sure you turn on drivers for your actual hardware; i.e., support for the proper network card and if you have a **SCSI** system the proper disk controller, and your particular sound card.

You can short circuit this whole procedure by obtaining a `.config` file that should work for most common hardware from **http:/www.coopj.com/LDD** with a name like `config-2.6.31_x86_64`. In this template we turn on the most common network cards etc and pick the options that will provide kernels that can handle the exercises we provide.

For detailed guidance on configuring kernels an invaluable resource is ***Linux Kernel in a Nutshell***, by Greg Kroah-Hartman, pub. O'Reilly, 2006, the full text of which is available at **http://www.kroah.com/lkn/**.

In order to compile modules against your kernel source you need more than just a proper `.config` file. Short of running a compilation first, doing `make prepare` or `make oldconfig` will take care of doing the setup for external module compilation, such as making symbolic links to the right architecture.

Step 4: Configure your boot loader (grub or lilo)

Before you can reboot, you'll need to reconfigure your boot loader to support the new kernel choice. Use either **grub** or **lilo**; you can't use both as they wipe each other out.

If you are using **grub**, you'll need to add a section to the configuration file (either **/boot/grub /grub.conf** or **/boot/grub/menu.lst** depending on your distribution) like:

```
title Linux (2.6.31)
        root (hd0,0)
        kernel /vmlinuz-2.6.31 ro root=LABEL=/
        initrd /initrd-2.6.31.img
```

which says the kernel itself is the first partition on the first hard disk (probably mounted as **/boot**) and the root filesystem will be found on partition with the label /. (Adjust partitions and labels as needed.)

Step 5: Compiling and installing the new kernel.

This involves:

- Making the **compressed** kernel (**bzImage**) and copying it over to the **/boot** directory with a good name. (On non-**x86** architectures, the kernel may be uncompressed.)
- Copying over the **System.map** file which is used to resolve kernel addresses mostly for logging and debugging purposes.
- Making modules and installing them under **/lib/modules/kernel-version/**.
- Saving the kernel configuration for future reference.
- Constructing a new **initrd** or **initramfs** image and copying it to the **/boot** directory.
- Updating your **grub** or **lilo** configuration.

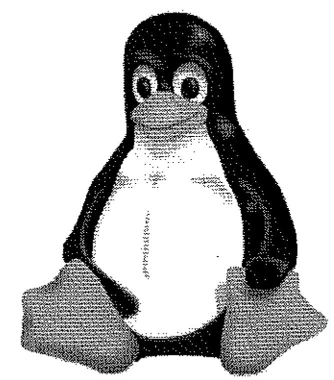
There is a script (**/sbin/installkernel**) on most distributions which can do these steps for you, and there is also an **install** target for **make**, but we prefer to use our own script over the canned one as it requires fewer arguments and is less rigid. This script is available in the solutions under the name **DO_KERNEL.sh**.

Thus if you want to use the canned configuration, you can do everything in this manner:

```
$ tar jxvf <pathto>linux-2.6.31.tar.bz2
$ cp <pathto>config-2.6.31_x86_64 linux-2.6.31/.config
$ cd linux-2.6.31
$ <pathto>DO_KERNEL.sh
```

Chapter 6

Kernel Features



We'll profile the major components of the kernel, such as process and memory management, the handling of filesystems, device management and networking. We'll consider the differences between user and kernel modes. We'll consider the important task structure and review scheduling algorithms. Finally we'll consider the differences between when the kernel is in process context and when it is not.

6.1 Components of the Kernel

Process Management

- Creating and destroying processes.
- Input and output to processes.

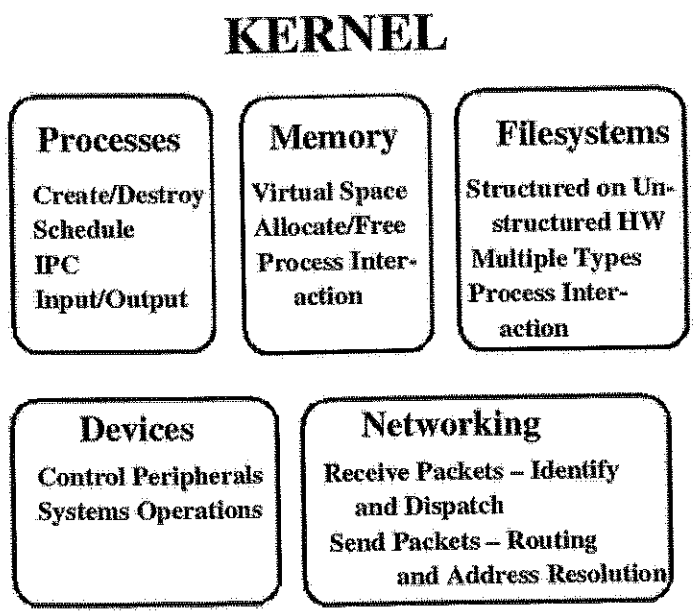


Figure 6.1: **Main kernel tasks**

- Inter-process communication (IPC) and signals and pipes.
- Scheduling.

Memory Management

- Build up a virtual addressing space for all processes.
- Allocating and freeing up memory.
- Process interaction with memory.

Filesystems

- Build structured filesystems on top of unstructured hardware.
- Use multiple filesystem types.
- Process interaction with filesystems.

Device Management

- Systems operations map to physical devices.
- **device drivers** control operations for virtually every peripheral and hardware component.

Networking

- Networking operations are not process specific; must be handled by the operating system.
- Incoming packets are asynchronous; must be collected, identified, dispatched.
- Processes must be put to sleep and wake for network data.
- The kernel also has to address routing and address resolution issues.

6.2 User-Space vs. Kernel-Space

Execution modes

user mode

Applications and daemons execute with limited privileges. (**Ring 3** on **x86**.) This is true even if the application has **root** privileges.

kernel mode

Kernel has direct, privileged access to hardware and memory. (**Ring 0** on **x86**.) Drivers (and modules) have kernel privileges.

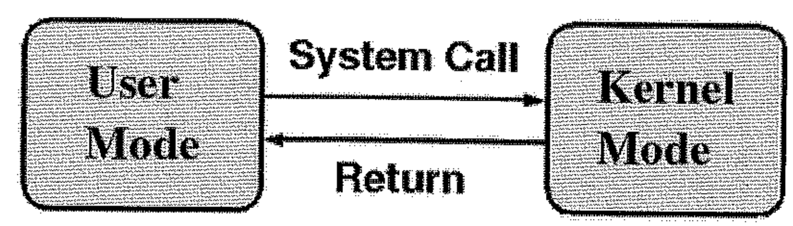


Figure 6.2: **User and kernel space**

Execution is transferred from user mode (space) to kernel mode (space) through system calls (which are implemented using synchronous interrupts, or exceptions) and hardware interrupts (or asynchronous interrupts).

The mode is a state of **each CPU** in a multi-processor system rather than the kernel itself, as each processor may be in a different execution mode.

- When running **virtualization** kernels the **hypervisor** (**Xen** for example) runs in Ring 0, while the guest (client) kernels may run in Ring 0 or Ring 1 depending on the type of virtualization.
- If it is Ring 1 a certain amount of trickery and/or emulation is required to accomplish this.

6.3 Scheduling Algorithms and Task Structures

Scheduling is arguably the most important work the kernel does. The main code for this is located in **/usr/src/linux/include/linux/sched.h** and **/usr/src/linux/kernel/sched.c**.

Tasks constantly switch back and forth between kernel mode and user mode (where they have lesser privileges.) Scheduling doesn't directly control these mode switches, but it does have to handle the context switching between different tasks.

Under **Linux**, a task by itself can not **preempt** a current running task and take over; it must wait its turn for a *time-slice*. However, the scheduler can preempt one task to allow another to run.

Tasks run until one of the following occurs:

- They need to wait for some system event to complete (such as reading a file.)
- The amount of time in their time-slice expires.
- The scheduler is invoked and finds a more deserving task to run.

Additionally, the 2.6 kernel has compile-time options for a **preemptible** kernel; when configured this way the kernel can behave much like a multi-processor system, lower latency, and preempt code even when it is in kernel mode. Thus all code which can be preempted this way must be fully re-entrant.

A task's `task_struct` is the critical data structure under **Linux**, and contains all information the kernel knows about a task, and everything it needs to switch it in and out. It is sometimes called a **process descriptor**. It is defined in **/usr/src/linux/include/linux/sched.h**.

The data structure of the current task (on the current CPU) can be referred to with the `current` macro; e.g., `current->tgid` is the current process ID and `current->pid` is the current task ID. (These can differ; for a multiple-threaded task each thread shares the same process ID but has a unique task (thread) ID.) This data structure contains information about signal handling, memory areas used by the process, parent and children tasks, etc.

Within the kernel schedulable processes (or more precisely tasks, or threads) that run either in user or kernel space are identified by pointers to a `task_struct`, not by a **pid**. For kernels earlier than 2.6.24 one can always obtain such a pointer from a **pid** with:

```
struct task_struct *find_task_by_vpid(int pid);
```

(Note that in a multi-threaded process this macro will locate the master thread, whose process identifier and thread identifier match.) However, later kernels do not export this macro to modules and one can use slightly more convoluted nested macros as in:

```
struct task_struct *t = pid_task(find_vpid(pid),PIDTYPE_PID);
```

One generally doesn't need to obtain this information from a module and race conditions can be a problem as **pid**'s can change during the lifetime of a process.

While the `schedule()` function may be called directly (from kernel code, not user code), it is more likely reached through an indirect call such as: when the current task goes to sleep and is placed

onto a **wait queue**; when a system call returns; just before a task returns to user mode from kernel mode; or after an interrupt is handled.

When the scheduler runs it determines which task should occupy the CPU, and if it is a different task than the current one, arranges the context switch. It tries to keep tasks on the same CPU (to minimize cache thrashing). When a new task is chosen to run, the state of the current task is saved in its `task_struct` and then the new task is switched in and made the current one.

The scheduler used before the 2.6.23 kernel was called the **O(1)** scheduler; the time required to make a scheduling decision was independent of the number of running tasks. It was designed to scale particularly well with **SMP** systems and those with many tasks. Separate queues were maintained for each CPU; these queues were kept in a priority-ordered fashion, so rather than having to tediously search through all tasks for the right task to run, the decision could be made through a quick bit-map consultation.

The 2.6.23 kernel saw replacement of the **O(1)** scheduler with an entirely new algorithm, which drives the **CFS** (**C**ompletely **F**air **S**cheduler) scheduler.

The completely fair time is the amount of time the task has been waiting to run, divided by the number of running tasks (with some weighting for varying priorities.) This time is compared with the actual time the task has received to determine the next task.

CFS also includes hierarchical **scheduler modules**, each of which can be called in turn.

6.4 Process Context

When the kernel executes code it always has full kernel privileges, and is obviously in kernel mode. However, there are distinct contexts it can be in.

In **process context** the kernel is executing code on behalf of a process. Most likely, a **system call** has been invoked and caused entry into the kernel, at one of a finite number of entry points. Examples would be:

- An application (or daemon) has issued a `read()` or `write()` request, either on a special device file, such as a serial port, or on a normal file residing on hardware to which the kernel has access to.
- A request for memory has been made from user-space. Once again only the kernel can handle such a request.
- A user process has made a system call like `getpriority()` to examine, or set, its priority, or `getpid()` to find out its process ID.

When not in process context, the kernel is not working on behalf of any particular user process. Examples would be:

- The kernel is servicing an **interrupt**. Requests to do so arrive on an **IRQ** line, usually in response to data arriving or being ready to send. For example a mouse click generates three or four interrupts. It is up to the kernel to decide what process may desire the data and it may have to wake it up to process the data.

- The kernel is executing a task which has been scheduled to run at either a specific time or when convenient. Such a function may be queued up through a kernel timer, a tasklet or another kind of **softirq**.
- The kernel is initializing, shutting down, or running the scheduler.

At such times the process context is not defined; although references to the `current task_struct` may not yield obvious errors, they are meaningless. Sometimes this situation is called **interrupt** context, but as we have seen it can arise even when no interrupts are involved.

The kernel context has a **lighter weight** than that of a user process; swapping in and out between kernel threads is significantly easier and faster than it is for full weight processes.

Recent kernels make more use of so-called **kernel processes**, which are much like user processes but which are run directly by the kernel.

Such pseudo-processes are used to execute management threads, such as the ones maintaining the buffer and page caches, which have to synchronize the contents of files on disk with memory areas. Other examples are the **ksoftirqd** and **kjournald** threads; do ps aux and look at processes whose names are surrounded by square brackets.

These processes are scheduled like normal processes and are allowed to sleep. However, they do not have a true process context and thus can not transfer data back and forth with user-space. In fact they all share the same memory space and switching between them is relatively fast.

You can examine what context you are in with the macros:

```
in_irq();       /* in hardware interrupt context     */
in_softirq();   /* in software interrupt context (bh)*/
in_interrupt(); /* in either hard/soft irq context   */
in_atomic();    /* in preemption-disabled context    */
```

Sleeping is disallowed if any of these macros evaluate as true.

6.5 Labs

Lab 1: Using strace.

strace is used to trace system calls and signals. In the simplest case you would do:

```
strace [options] command [arguments]
```

Each system call, its arguments and return value are printed. According to the man page:

"Arguments are printed in symbolic form with a passion."

and indeed they are. There are a lot of options; read the man page!

As an example, try

```
strace ls -lRF / 2>&1 | less
```


You need the complicated redirection because **strace** puts its output on `stderr` unless you use the `-o` option to redirect it to a file.

While this is running (paused under **less**), you can examine more details about the **ls** process, by examining the **/proc** filesystem. Do

```
ps aux | grep ls
```

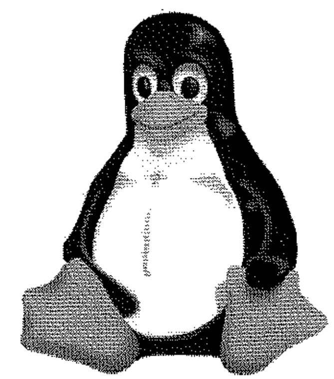
to find out the process ID associated with the process. Then you can look at the pseudo-files

```
/proc/<pid>/fd/*
```

to see how the file descriptors are mapped to the underlying files (or pipes), etc.

Chapter 7

Kernel Style and General Considerations



We'll discuss what style kernel code should be written in to heighten its chances of inclusion in the main kernel source tree. We'll show how to make and use a kernel **patch**. We'll discuss using the **sparse** analysis tool. We'll consider how **Linux** uses a unified method to deal with **linked lists**. Then we'll consider various practices that should be followed to make code that is portable and future-oriented, in particular with regards to using already in-place kernel methods, word size and endianness, and making sure code works on multi-processor and high memory systems. We'll note that security aspects should be kept in mind at all times. Finally, we'll talk about keeping kernel and user-space headers separate.

7.1 Coding Style

The style in which kernel code is done will have a major influence on whether any patch, or driver, you write makes its way into the official kernel tree. If some basic conventions are not followed, it is likely the kernel maintainers will not even consider it.

The official document on this topic can be found in the kernel documentation, at **/usr/src/linux /Documentation/CodingStyle**. It contains some general precepts as well as specific rules; we won't try and summarize it in detail. But a few points are worth mentioning:

- Code should be rationally indented; 8 characters is the preferred value for tab stops. (**Note:** in this material we sometimes use smaller indentations because of page-width limitations.)
- The basic style can be obtained by using the script **/usr/src/linux/scripts/Lindent**, which boils down to the command:

  ```
  indent -npro -kr -i8 -ts8 -sob -l80 -ss -ncs -cp1 "$@"
  ```

- Namespace pollution should be avoided; global variables should have descriptive names. Mixed-case names are discouraged; lower case with underscores is common; e.g., instead of `MyCriticalVariable`, use `my_critical_variable`. In general names should be short.
- Avoid complex functions with multiple purposes and lots of local variables. Helper functions should be used (and can be in-lined by the compiler if efficiency is critical.) The general rule is: short and sweet, one facility per function.
- Avoid cute obfuscation and over-condensation. We've all seen compact C-code where multiple statements are packed into one line. Clarity is more important than brevity. Clear code is much easier to maintain; remember this is open source and the number of eyeballs on your code will be very large.
- Comments should be economical and not overdone. The main purpose should be to explain non-obvious steps; clean code shouldn't require many comments.

It is possible to customize your **emacs** initialization files to provide much of this style automatically. You can also run source code through the `indent` program, and through the incredibly large choice of options, make it have almost any style you want.

Macros of the form:

```
#define my_macro(x,y)   \
do {                    \
    for (;;){           \
       if (x>y)         \
           break;       \
       schedule();      \
    }                   \
} while (0)
```


often confuse people who want to remove the `do {...}while (0)` construction, but they are convenient when the macro includes multiple statements.

7.2 kernel-doc

The **kernel-doc** format is often used in the **Linux** kernel to embed comments in the source. In addition to providing a uniform standard, the **kernel-doc** utilities also provide easy to use tools for extracting this information in convenient formats.

A simple example from **/usr/src/linux/Documentation/kernel-doc-nano-HOWTO.txt** suffices to demonstrate the format that must be used:

```
/**
 * foobar() - short function description of foobar
 * @arg1:      Describe the first argument to foobar.
 * @arg2:      Describe the second argument to foobar.
 *             One can provide multiple line descriptions
 *             for arguments.
 *
 * A longer description, with more discussion of the function foobar()
 * that might be useful to those using or modifying it.  Begins with
 * empty comment line, and may include additional embedded empty
 * comment lines.
 *
 * The longer description can have multiple paragraphs.
 **/
```

Such comment blocks should be placed just before the function or data structure being documented, and the first line must be on a single line, with no lines before the argument lines.

Over-commenting in the **Linux** kernel is definitely discouraged, but any function (or data element) which is lent to modules through `EXPORT_SYMBOL()`, or that is not declared as `static` and thus is global in scope, is a good candidate.

Extraction of the documentation is done through the use of **/usr/src/linux/scripts/kernel-doc**; running without any arguments shows its use:

```
$ /usr/src/linux/scripts/kernel-doc
Usage: /usr/src/linux/scripts/kernel-doc [ -v ]
         [ -docbook | -html | -text | -man ]
         [ -function funcname [ -function funcname ...] ]
         [ -nofunction funcname [ -nofunction funcname ...] ]
         c source file(s) > outputfile
```

7.3 Using Generic Kernel Routines and Methods

Avoid reinventing the wheel. There are many standard methods in place within the kernel and they should be used where possible. Two examples:

- Linked lists, in particular, doubly-linked (sometimes circular) lists (those with `next`, `prev` pointers) should use the routines defined in **/usr/src/linux/include/linux/list.h**. These handle all major tasks, such as insertion and deletion and walking through a list.
- Testing and setting bits in flags should be done safely with the functions described in **/usr/src/linux/arch/x86/include/asm/bitops.h**.

There is a lot of duplication in the **Linux** kernel, particularly among device drivers. For instance many network drivers started from one or a few early ones and then morphed. You may find drivers which differ in just a few lines of code.

Furthermore, because of the rather unique way in which the **Linux** code base has evolved, there has been quite a bit of parallel evolution.

While this kind of re-use is a **good thing** and is sometimes encouraged, from time to time natural selection is enforced; if one method is superior, code containing the other method is converted. Don't sidestep known facilities without good reason. Your code will be rejected.

In a commercial product, such duplication might be avoided by more use of library-type routines, or by writing one driver that can handle multiple hardware devices with conditional branches. As **Linux** has matured more of this has happened.

7.4 Making a Kernel Patch

A **patch** is merely a text file that contains the differences between your modified source and the baseline source. It is produced with the **diff** program and applied with the **patch** utility. Making a patch file is trivial; doing it properly so that people can use it readily requires a little care.

To produce a patch, suppose the original source is in `/usr/src/linux`, and suppose your modified source is in `/usr/src/linux_hacked`. You can produce the patch with the command:

```
cd /usr/src
diff -Nur linux linux_hacked > hacked_patch
```

where the `-u` option specifies **unified** format, the `-r` option forces **diff** to recurse through subdirectories, and the `-N` option says to consider files present in one tree and not the other.

You can then distribute this to anyone who has the original source and then they can apply the patch simply by doing:

```
cd /usr/src/linux
patch -p1 < hacked_patch
```

However, for this to work properly you have to be careful, considering the following points:

- The original source (that under `/usr/src/linux`) should be truly pristine as it was when it was first unpacked. In any accompanying documentation you should say exactly which source you are patching. It is acceptable to submit a patch of a distribution-supplied source, but clearly give the name and pointer to the source you use. If this is for the kernel mailing list, use **only** authentic unpatched original sources from **http://www.kernel.org**

- Your modified source should be cleaned up before producing the patch, perhaps using `make mrproper`. Only files that have actually changed should be included, and annoying changes in items such as white space should be eliminated.

The above procedure is complete enough, but it can be time consuming as it requires diffing the whole kernel source even if there are just a few changes, and it might require keeping up to 3 or 4 versions of the source around.

One good trick is to make **hard links** instead of copies between the parts of the kernel you are not changing and the originals, and diffing will go very fast. To do this you would do:

```
cd /usr/src
cp -al linux linux_work
```

Then suppose you want to change only one file, say `kernel/sys.c`. You would do:

```
cd linux_work/kernel
rm sys.c
cp /usr/src/linux/kernel/sys.c .
```

Removing `sys.c` removes only the hard link in the current directory, not the original file. Now you have two entire directory trees that are hard linked together, except for the one file that you are going to work on, and the diffing will be very fast indeed. (Note that the **rm** and **cp** steps are unnecessary when you are using some text editors, such as **emacs** to update the files; however with **vi** it is necessary.)

For a detailed description of the exact format for patches submitted to the linux kernel mailing list, see **http://linux.yyz.us/patch-format.html**, **http://www.zip.com.au/ akpm/linux/patches /stuff/tpp.txt**, and **/usr/src/linux/Documentation/SubmittingPatches**.

Before submitting a patch one should run the script **/usr/src/linux/scripts/checkpatch.pl** on it and clean up any warnings and errors that result.

7.5 sparse

sparse is a general purpose **C**-language parsing and analysis tool, originally written by Linus Torvalds. (**Sparse** stands for **S**emantic **Parser**.) By attaching various back-ends onto it it can serve many purposes. For example, if one were to attach a code-generation back-end, a compiler would be the result.

For the kernel, however, the back-end used is analysis code designed to scream about certain kinds of errors (such as type mismatches), the list of which has been growing since **sparse** first appeared.

Here's an example of two (possible) errors **sparse** can pick up on:

```
static ssize_t mycdrv_read (struct file *file, char *buf, size_t count, loff_t * ppos);
....
remove_proc_entry ("pre", 0);
```

In the first line buf points to a user-space buffer, and should be tagged with the __user attribute. In the second line one should be using NULL for the null pointer instead of a value of 0. Thus the corrected code would be:

```
static ssize_t mycdrv_read (struct file *file, char __user *buf, size_t count, loff_t * ppos);
.....
remove_proc_entry ("pre", NULL);
```

Note that during normal compilation, no noise is made about these "errors" and the __user attribute is totally ignored; e.g., **sparse** is a syntax checker, not a compiler.

The official web page for **sparse** is **http://kernel.org/pub/linux/kernel/people/josh/sparse/** and the latest official release can be obtained from there. If you want to live on the edge, the latest development code snapshot can be obtained from **http://www.codemonkey.org.uk/projects/git-snapshots/sparse/**

After untarring and decompressing it, it can be compiled and installed with:

```
make PREFIX=/usr/local install
```

which will put the binary in /usr/local/bin; you can adjust the value of PREFIX as desired.

Invocation of **sparse** is rather simple. You just to do

```
make C=2 .....
```

when compiling modules or the kernel; A value of C=1 does just the specific module file; a value of C=2 does all files.

7.6 Using likely() and unlikely()

Kernel code has had a history of using many goto statements. This has often caused newcomers to shake their heads. For instance one might have something like:

```
    if (test1)
        goto flunk_test1;
test1_continue:
    if (test2)
        goto flunk_test2;
test2_continue:

.......

flunk_test1:
     .... do something 1 ....
     goto test1_continue
flunk_test2:
     .... do something 2 ....
     goto test2_continue
```


At first glance, this looks rather inefficient and can be difficult to follow; it would seem to be better to do:

```
if (test1){
 .... do something 1....
}
if (test2){
 .... do something ....
}
```

However, the second code example can be less efficient. Because the compiler works together with the processor on **branch prediction**, if the first example is written so that the **most likely case almost always falls through** it will run quicker, as the code further away in the source is assumed less likely to be encountered.

Because the code can become confusing, especially to new examiners, such methods make the most sense when the code is very often executed and saving a few cycles is very important; it was once widely used in the scheduling routines for instance.

Beginning with **gcc** version 2.96, **Linux** aids such prediction through the likely() and unlikely() macros, defined in **/usr/src/linux/include/linux/compiler.h** in terms of the **gcc** macro, __builtin_expect(). When coded this way our example becomes:

```
if (unlikely(test1)){
 .... do something 1....
}
if (unlikely(test2)){
 .... do something ....
}
```

with the use of the likely() macro quite similar. The code is now easier to understand **and** gets the benefits of branch prediction.

7.7 Linked Lists

Linked lists of data structures are very common in the **Linux** kernel, as they are in most major software projects. They may be singly or doubly linked, and they may have ends or be cyclical.

A typical doubly linked list implementation would define a data structure with embedded next and prev pointers, such as in:

```
struct my_struct {
    struct my_struct *next, *prev;
    int val;
    char *my_data;
    ....
}
```

If the list is not cyclical it terminates with NULL for next at one end, and prev at the other. If it is cyclical it joins on itself, and traversing the list involves stepping through it until you arrive at the starting element again.

While there is nothing wrong in principle with direct implementation of such a linked list, including a new one is likely a good way to get your code rejected. **Linux** kernel developers have standardized on one clever implementation and it should be adopted in new code.

One advantage is that with a conventional implementation, one has to provide a battery of functions for insertion, removal, splicing, deletion, joining and other manipulations of the linked list. This is because every one is somewhat different because of the differing nature of the data structures. With the **Linux** standardized implementation, the same functions are used no matter what the linked data structures are made of

A second advantage is the one that one is using well-tested, safe functions, which are generalized to work on all architectures, and on which gradual refinements, enhancements, and improvements are made.

The various functions and structures involved are detailed in **/usr/src/linux/include/linux/list.h**. The elementary data structure is

```
struct list_head {
   struct list_head *next, *prev;
};
```

Any and each **node** in the linked list can be used to start traversal; hence the unusual name `list_head`. To use this facility you need to place a `list_head` structure inside the structure you are linking. For example,

```
struct my_struct{
   struct list_head list;
   int val;
   char *my_data;
   ....
}
```

The critical point to understand is that the **next, prev** pointers in the `list_head` structure **do not** point to the data structures in which the `list_head` is embedded. Instead, they point to the `list_head` field **within** those structures. To retrieve a pointer to the data structure itself, one needs to use the `list_entry()` macro detailed shortly.

The **list head** must be initialised prior to use. This can be done statically at compile time as:

```
LIST_HEAD(my_list);
```

or

```
struct my_struct me = {
   .list = LIST_HEAD_INIT (me.list);
   .val = 0;
   .my_data = NULL;
}
```

or at run time as:


```
struct my_struct *me = kmalloc (sizeof(struct my_struct), GFP_KERNEL);
me->val = 0;
me->my_data = NULL;
INIT_LIST_HEAD (&me->list);
```

```
struct list_head my_list;
INIT_LIST_HEAD(&my_list);
```

The main functions (some are macros) involved in manipulating doubly-linked lists are:

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

void list_add (struct list_head *new, struct list_head *head);
void list_add_tail (struct list_head *new, struct list_head *head);
void list_del (struct list_head *entry);
int list_empty (struct list_head *head);
void list_splice (struct list_head *list, struct list_head *head);
```

`list_add()` inserts an element pointed to by its first argument after that pointed to by its second argument.

`list_add_tail()` inserts an element pointed to by its first argument at the end of the list pointed to by its second argument.

`list_del()` removes the element pointed to from its linked list. One must still deallocate any memory associated with the linked data structure.

`list_empty()` checks if the pointed to list is empty.

`list_splice()` joins two lists together, where the first argument points to the new list and the second tells where to insert it.

The following macros are used to work through a linked list:

```
list_entry (ptr, type, member);
list_for_each (pos, head);
list_for_each_prev (pos, head);
list_for_each_safe (pos, n, head);
```

`list_entry()` returns a pointer to the data structure of the type indicated in its second argument, from the list whose head is pointed to by the first argument, and of which the member we desire is given by the third argument.

`list_for_each()` is used to iterate over the list pointed to by its second argument, performing operations on its first argument. `list_for_each_prev()` iterates over the list backward.

`list_for_each_safe()` handles the case where one is removing the list entry; the second argument is a pointer to a `struct list_head` that is used for temporary storage.

A variant is to use the `list_for_each_entry (pos, head, member)` convenience macro (where `member` is the name of the list structure within the structure) so that the two following code fragments accomplish the same thing:

```
LIST_HEAD (h);
struct list_head *l;
struct my_s *s;
list_for_each (l, &h) {
    s = list_entry (l, struct my_s, list);
    ....
}
```

or

```
LIST_HEAD (h);
struct my_s *s;
list_for_each_entry (s, &h, list) {
    ....
}
```

There are some other functions and macros, and a lot of documentation in the header file. Here's a code fragment showing how to set up a linked list, add elements to it, and traverse it, running a function on its items:

```
LIST_HEAD (my_list);

struct my_entry
{
    struct list_head list;
    int intvar;
    char strvar[20];
};

static void mylist_init (void)
{
    struct my_entry *me;
    int j;
    for (j = 0; j < NENTRY; j++) {
        me = kmalloc (sizeof (struct my_entry), GFP_KERNEL);
        me->intvar = j;
        sprintf (me->strvar, "My_%d", j + 1);
        list_add (&me->list, &my_list);
    }
}

static int walk_list (void)
{
    int j = 0;
    struct list_head *l;

    if (list_empty (&my_list))
        return 0;

    list_for_each (l, &my_list) {
        struct my_entry *me = list_entry (l, struct my_entry, list);
        foobar (&me->intvar);
        j++;
```



```
    }
    return j;
}
```

7.8 Writing Portable Code - 32/64-bit, Endianness

Linux has been ported to more platforms than any other operating system. Even if the code you are writing is intended for one particular kind of hardware, as far as possible your code should be hardware-independent. There are at least two reasons for this:

- There is an ongoing effort to keep code as architecture-clean as possible. If platform-dependent code proliferates, even where it is clearly intended only to be used on one kind of hardware, it becomes difficult to root out those locations where the code really could be hardware-independent, but it isn't either through the process of evolution or sloppy design.
- You can never be sure what platform the kernel (and your code) may be run on in the future. For instance your device may at some point be hooked up to an **IA-64** motherboard, rather than the **x86** one you were thinking of when you developed the driver for it.

Thus you should never make assumptions about parameters such as the page size, length of an address, etc. Use the general variable types and definitions used in the kernel; i.e., use `offset_t` for a file offset, not an `unsigned long`. Where the actual byte-length of a variable is important and rigid, one should use the built-in kernel types, like u32 etc.

Assumptions about **endianness** (big-endian vs. little-endian) should be avoided. Use the available kernel functions which are conscious of endianness, such as those that manipulate **PCI** configuration registers, and various network facilities.

Truly platform-dependent code should be confined to the appropriate `arch` directory.

7.9 Writing for SMP

While symmetric multiprocessor (**SMP**) machines are still relatively rare, hyper-threaded and multi-core CPUs which behave much like multiprocessor systems are now quiet common. Your code is likely to be run on **SMP** machines even if it deals with a low-performance kind of hardware or facility. Thus you must always think about questions like:

- What if more than one processor is running the code at the same time?
- Are there global variables that require synchronization?
- Can an interrupt be dealt with on one CPU while another is handling a read or a write to the device, and thereby corrupt data?

Avoiding **race conditions** requires good design and careful thinking of all the possibilities, and the various synchronization facilities must be utilized, such as **spinlocks, semaphores**, etc..

On single CPU systems some of the synchronization directives may become no-ops, but you still have to employ them. On an **SMP** system, that simple mouse driver you wrote might wind up paralyzing the system.

Obviously, it is not possible to test fully for **SMP** behaviour on a single processor system. But one minimal step you can do is to use an **SMP** kernel, and compile your code with **SMP** defined. Wherever it is humanly possible, try to test your code on an **SMP** system.

A good test is to turn on the **kernel preemption** configuration option on a single processor machine; many **SMP** bugs may be uncovered.

7.10 Writing for High Memory Systems

Don't assume that your driver will only be used on systems without much memory. Wherever possible think about what will happen if the system has a lot of RAM.

Be careful about things like the number of bits in an address or a file offset, or doing anything that scales with the amount of available RAM.

7.11 Keeping Security in Mind

Major security holes can be caused by even minor errors in kernel code. Code should be reviewed early and often with security in mind.

User-space parameters should not be trusted by default. Drivers (and other facilities) should check permissions and use the **capability** functions to check whether or not anything that is requested is permissible.

Care should be taken with respect to integer type mismatches, type casts, etc., so nothing unforeseen takes place. Resource usage should be limited to avoid **Denial of Service** attacks.

7.12 Mixing User- and Kernel-Space Headers

Applications and libraries in user-space, and kernel code (modular or not), require the inclusion of **headers** (files with a `.h` extension.)

Because the kernel is an isolated universe, it can't use user-space headers, it can't link to user-space libraries etc. All kernel code uses the headers lying under `/usr/src/linux/include` or `/usr/src/arch/../include` (assuming the kernel source is at `/usr/src/linux`.)

This is very different than the way it is done on other operating systems so it takes **Linux** converts some time to get used to. The upshot is kernel files must **never, never** include user-space headers. This means you can't have anything like

```
#include <stdio.h>
#include <sys/types.h>
#include <bits/errno.h>
```


Note no header files can be found directly in the `/usr/src/linux/include` directory, and almost all header files are in the `linux` or `asm` directories.

(**Note:** some recent kernels have broken this rule in order to better interact with user-space debuggers. This seems to be the only acceptable violation.)

Applications and libraries, on the other hand, use headers which usually reside under `/usr/include` and a few other places. This is associated with a second rule: User code must **almost never** include kernel-space headers directly. While it is true that a header file like `/usr/include/unistd.h` will eventually wind up including `/usr/src/linux/include/linux/unistd.h`, it should be done indirectly, not directly.

There are some exceptions to this rule; they are always in cases where some direct interface with the kernel is required, generally a very non-portable one. Examples are when making systems calls directly from programs without passing through **libc** on the way, or creating processes and/or threads with the low-level `clone()` call.

Another consideration is the fact that user-space libraries and applications may need to know information about the kernel and have to interface with it through **system calls**. Unfortunately, one easy method is for user-space code to include various kernel-space headers. However, if the kernel is changed so will these headers, so there is the potential danger that there will be a collision between the headers with which the kernel and application were compiled, or that an application (or library) may require re-compilation.

To avoid this, distributions include a version of the kernel headers that is packaged with **glibc** and were in effect at the time the system and the libraries were compiled and assembled, placed directly under `/usr/include`. This can cause some inconveniences when compiling kernel code, but is generally a preferable solution.

The side effect of the above considerations is that since the default behaviour of the compiler is to search `/usr/include` before `/usr/src/include`, when you compile **kernel code** the location of the kernel headers is explicitly specified with the `-I` option, and `-nostdinc` is supplied as a compiler option to make sure standard headers are not picked up accidentally.

7.13 Labs

Lab 1: Linked Lists

Write a module that sets up a doubly-linked circular list of data structures. The data structure can be as simple as an integer variable.

Test inserting and deleting elements in the list.

Walk through the list (using `list_entry()`) and print out values to make sure the insertion and deletion processes are working.

Lab 2: Finding Tainted Modules

All modules loaded on the system are linked in a list that can be accessed from any module:

```
struct module {
....
struct list_head modules;
....
char name[MODULE_NAME_LEN];
....
unsigned int taints;
....
}
```

Write a module that walks through this linked list and prints out the value of taints and any other values of interest. (The module structure is defined in **/usr/src/linux/include/linux/module.h**.)

You can begin from THIS_MODULE.

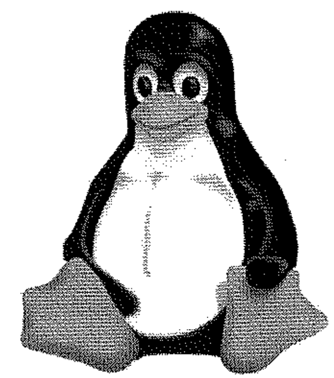
Lab 3: Finding Errors With Sparse

We give you a minimal module that compiles cleanly, but has at least two errors that show up with the use of **sparse**.

Install **sparse** according to the description given earlier and correct the errors.

Chapter 8

Interrupts and Exceptions



We'll take a detailed look at how the **Linux** kernel handles synchronous interrupts (exceptions) and asynchronous interrupts. We'll consider message-signalled interrupts (**MSI**). We'll show how to enable/disable interrupts. We'll have a discussion of what you can and can not do when in interrupt context. We'll consider the main data structures associated with interrupts and show how to install an interrupt handler, or service routine. Finally we'll discuss in detail what has to be done in the top and bottom halves of such functions.

8.1 What are Interrupts and Exceptions?

An **interrupt** alters (interrupts) the instruction sequence followed by a processor. It is always connected with an electrical signal stemming from either inside or outside the processor.

When an interrupt arrives the kernel must suspend the thread it is currently executing, deal with it by invoking one or more **service routines (ISR)** (or **handlers**) assigned to the specific interrupt, and then return to the suspended thread, or service another interrupt.

Under **Linux**, interrupts should never be lost; the service routines may be delayed according to various locking mechanisms and priorities, but will be invoked eventually. However, interrupts are not queued up; only one interrupt of a given type will be serviced, although if another interrupt of the same type arrives while it is being serviced, it too will be serviced in turn.

There are two distinct kinds of interrupts:

- **Synchronous** interrupts, often called **exceptions**, are generated by the CPU.
- **Asynchronous** interrupts, often just called **interrupts**, are generated by other hardware devices, and are generally fed through the **APIC** (**A**dvanced **P**rogrammable **I**nterrupt **C**ontroller.)

Exceptions may be caused by run time errors such as division by zero, or by special conditions such as a page fault. They may also be caused by certain instructions, such as the one a system call makes to request the CPU enter kernel mode to service a request from user-land. They may occur in or out of process context. They often cause a signal to be sent to one or more processes.

Interrupts generally arise from relatively random events such as a mouse click, keyboard press, a packet of data arriving on a network card, etc, or more regular events such as a timer interrupt. They are never associated with a process context.

Interrupts are very similar to signals; one might say interrupts are hardware signals, or signals are software interrupts. The general lessons of efficient and safe signal handling apply equally well to interrupts.

Interrupt handling is one of the most difficult tasks incurred by the kernel. It requires careful design to avoid race conditions and problems with non-reentrant code.

Under **Linux** interrupts may be shared, and when an interrupt is shared, all handlers for that interrupt must agree to share. Each of them will receive the interrupt in turn; i.e., there is no **consumption** of the interrupt by one of the handlers.

8.2 Exceptions

Exceptions fall into two categories:

Processor-detected exceptions are generated when the CPU senses a condition during instruction execution. These can be of three types, according to the value of the **eip** register on the kernel mode stack:

- **Faults**: the register contains the address of the instruction that induced the fault; when the exception service routine completes, execution will be resumed from that instruction if the handler is able to deal with the anomalous condition that produced the exception, such as a page fault.
- **Traps**: the register contains the address of the instruction to be executed after the one that induced the trap. Traps are mainly used for debugging and tracing methods and there is no need to repeat the instruction that caused the trap.
- **Aborts**: the register may not contain a meaningful value. There may be a hardware failure or a invalid value in system tables. The abort handler will terminate the affected process.

Programmed exceptions are requested by the process through **int** or **int3** instructions (on **x86**), such as when invoking a system call. They may also be triggered by the **into** instruction which checks for overflow, or the **bound** instruction, which checks on address bound, when the checked condition is false. Programmed exceptions are handled just like **traps** and are sometimes called **software interrupts**.

On 32-bit **x86** CPUs there are up to 32 exceptions possible, numbered from 0 to 31. The exact number depends on the processor. The signal listed in the following table is usually sent to the process which triggered the exception.

Table 8.1: **32-bit x86 exceptions**

#	Exception/ Service Routine	Type	Signal	Meaning
0	Divide Error `divide\_error()`	fault	`SIGPE`	Attempted division by 0.
1	Debug `debug()`	trap or fault	`SIGTRAP`	Used by debugging and tracing programs.
2	NMI `nmi()`	None	None	Reserved for nonmaskable interrupts that use the NMI pin.
3	Breakpoint `int3()`	trap	`SIGTRAP`	Debugger has inserted a **int3** instruction (breakpoint).
4	Overflow `overflow()`	trap	`SIGSEGV`	An **into** instruction has been executed and overflow detected.
5	Bounds Check `bounds()`	fault	`SIGSEGV`	A **bounds** instruction has been executed and the operand is outside of valid bounds.
6	Invalid opcode `invalid\_op()`	fault	`SIGILL`	Bad opcode (part of the CPU instruction that selects the operation.)
7	Device not available `device\_not\_available()`	fault	`SIGSEGV`	A floating point or MMX instruction executed with the TS flag of cr0 set.
8	Double fault `double\_fault()`	abort	`SIGSEGV`	An exception detected while trying to handle a prior one, and for some reason they can't be handled in turn.
9	Co-processor segment overrun `coprocessor\_segment\_overrun()`	abort	`SIGFPE`	Problem with a math coprocessor, like a x387 chip.

10	Invalid TSS invalid_tss()	fault	SIGSEGV	Attempted context switch to a process with an invalid TSS.
11	Segment not present segment_not_present()	fault	SIGBUS	Reference made to a segment not present in memory.
12	Stack Exception stack_segment()	fault	SIGBUS	An attempt was made to exceed the stack segment length, or the segment is not in memory.
13	General Protection general_protection()	fault	SIGSEGV	Protected mode has been violated.
14	Page fault page_fault()	fault	SIGSEGV	Page is not in memory, the page table entry is null, or some other paging problem.
15	Reserved	None	None	None
16	Floating Point Error co-processor_error()	fault	SIGFPE	Integrate FPU has an error condition such as overflow or division by 0.
17	Alignment Check alignment_check()	fault	SIGSEGV	Operand not correctly aligned, such as a long integer on an address not a multiple of 4.
18	Machine Check machine_check()	abort	None	Severe hardware problem, usually in the CPU, or memory.
19	SIMD error simd_coprocessor_error()	fault	SIGSEGV	Problems with executing SIMD (Single Instruction Multiple Data) math instruction.

Note that the actual handler function is prefixed with do_. Thus the page fault exception handler function is do_page_fault(), with the actual work of pointing to the real function being done in **/usr/src/linux/arch/x86/kernel/entry_32.S**.

8.3 Interrupts

There are two kind of asynchronous interrupts:

- **Maskable** interrupts are sent to the **INTR** microprocessor pin. They can be disabled by appropriate flags set in the **eflags** register.
- **Nonmaskable** interrupts are sent to the **NMI** microprocessor pin. They can not be disabled and when they occur there is usually a critical hardware failure.

Any device which issues interrupts has an **IRQ** (Interrupt **ReQ**uest) line, which is connected to an **APIC** (**A**dvanced **P**rogrammable **I**nterrupt **C**ircuit.)

On the **x86** architecture one has a **Local APIC** integrated into each CPU. Additionally one has an **I/O APIC** used through the system's peripheral buses.

The **I/O APIC** routes interrupts to individual **Local APICs**, according to a redirection table that it keeps.

The **Local APIC** constantly monitors the IRQ lines it is responsible for and when it finds a signal has been raised:

- Notes which **IRQ** is involved.
- Stores it in an I/O port it owns so it can be read on the data bus.
- Issues an interrupt by sending a signal to its INTR pin.
- When the CPU acknowledges the IRQ by writing back into a controller I/O port it clears the INTR pin.
- Goes back to waiting for a new interrupt to arrive.

A list of currently installed IRQ handlers can be obtained from the command cat **/proc/interrupts**, which gives something like:

```
          CPU0     CPU1     CPU2     CPU3
  0:       129        1        2        1   IO-APIC-edge      timer
  1:       722       12       40       32   IO-APIC-edge      i8042
  8:         0        0        0        1   IO-APIC-edge      rtc0
  9:         0        0        0        0   IO-APIC-fasteoi   acpi
 16:      1546      948      102       90   IO-APIC-fasteoi   uhci_hcd:usb3, pata_marvell, nvidia
 18:     16287       10       14    18141   IO-APIC-fasteoi   eth0, ehci_hcd:usb1, uhci_hcd:usb5,
                                                              uhci_hcd:usb8
 19:         2        1        1        0   IO-APIC-fasteoi   uhci_hcd:usb7, ohci1394
 21:         0        0        0        0   IO-APIC-fasteoi   uhci_hcd:usb4
 22:       534      151     4510     2284   IO-APIC-fasteoi   HDA Intel
 23:     44436    38789     4644     4122   IO-APIC-fasteoi   ehci_hcd:usb2, uhci_hcd:usb6
 28:         0        0        1        0   PCI-MSI-edge      eth1
 29:     17358     2567    18150    17191   PCI-MSI-edge      ahci
NMI:         0        0        0        0   Non-maskable interrupts
LOC:    255559    16184   147985   131046   Local timer interrupts
SPU:         0        0        0        0   Spurious interrupts
RES:      1849     1183     1520     1158   Rescheduling interrupts
CAL:       340      475      475      450   Function call interrupts
TLB:      1773     2734     1720     2689   TLB shootdowns
TRM:         0        0        0        0   Thermal event interrupts
THR:         0        0        0        0   Threshold APIC interrupts
ERR:         0
MIS:         0
```

Note that the numbers here are the number of times the interrupt line has fired since boot. Only currently installed handlers are listed. If a handler is unregistered (say through unloading a module) and then it or another handler is later re-registered, the number will not be zeroed in the process.

You will also notice two types of interrupts:

- **Level**-triggered interrupts respond to an electrical signal (generally a voltage) having a certain value.

- **Edge-triggered** interrupts respond to a change in electrical signal, which can be either up or down.

In principle one could miss a level-triggered interrupt if it is cleared somehow before the change in condition is noticed.

On **SMP** systems interrupts may be serviced on any available CPU (although affinities can be mandated), but only one CPU will handle an interrupt of a certain kind at the same time.

It is sometimes advantageous to set **IRQ**-affinity; to force particular interrupts to be dealt with only some subset of all the CPUs, rather than being distributed roughly equally.

This is done by accessing `/proc/irq/IRQ#/smp_affinity`. (Note on 64-bit platforms this hexadecimal entry has two 8 digit sets, separated by a comma.) One can not turn off all CPUs in the mask, and won't work if the physical **IRQ** controller doesn't have the capability to support an affinity selection.

The **irqbalance** daemon dynamically adjusts the **IRQ** affinity in response to system conditions. It takes into account performance (latency and cache coherence) and power consumption (keeping **CPU**s no more active than necessary when system load is light.)

There was also an in-kernel **IRQ**-balancing option, but this was deprecated and finally removed in kernel version 2.6.29 in favor of the user-space solution. Full documentation about the daemon method can be found at **http://www.irqbalance.org**.

8.4 MSI

In pre-**PCI-e** (PCI-express) buses interrupts are **line-based** and are now considered as legacy technology. The external pins that signal interrupts are wired separately from the bus main lines, producing **out of band** signalling.

PCI-e maintains compatibility with older software by emulating this legacy behaviour with **in-band** methods, but are still limited to only four lines and often require sharing of interrupts among devices.

The **PCI** 2.2 standard added a new mechanism known as **MSI** (for Message-Signalled Interrupts), which was further enhanced in the **PCI** 3.0 standard to become **MSI-X**, which is backward compatible with **MSI**.

Under **MSI** devices send 16-bit **messages** to specified memory addresses by sending an inbound memory write to the front side bus (**FSB**). The message value is opaque to the device but delivery generates an interrupt. The message is not acknowledged, and thus we get an edge-triggered interrupt.

Under **MSI** each device can use up to 32 addresses and thus interrupts, although the operating system may not be able to use them all. The address is the same for each message, but they are distinguished by modifying low bits of the message data.

Under **MSI-X** the messages become 32-bit and up to 2048 individual messages can be sent for each device. Each **MSI-X** interrupt uses a different address and data value (unlike in **MSI**).

There are important advantages of using message-signalled interrupts. First, the device no longer has to compete for a limited number of **IRQ** lines; thus there is no need to share. Interrupt latency is therefore potentially reduced and getting rid of sharing also makes behaviour more predictable and less variable.

MSI is optional for **PCI** 2.3 compliant devices and mandatory for PCI-Express devices. Support for **MSI** and **MSI-X** must be configured in the kernel to use them; only one standard can be used in a particular driver at a time.

Details about the **Linux** implementation are given in **/usr/src/linux/Documentation/MSI-HOWTO.txt**, which contains details about the **API** and enumerates important considerations.

8.5 Enabling/Disabling Interrupts

Sometimes it is useful for a driver to enable and disable interrupt reporting for an IRQ line. The functions for doing this are:

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

void disable_irq (int irq);
void disable_irq_nosync (int irq);
void enable_irq (int irq);
```

These actions are effective only for the CPU on which they are called; other processors continue to process the disabled interrupt.

Because the kernel automatically disables an interrupt before calling its service routine and enables it again when done, it makes no sense to use these functions from within the handler servicing a particular IRQ.

Calling `disable_irq()` ensures any presently executing interrupt handler completes before the disabling occurs, while `disable_irq_nosync()` will return instantly. While this is faster, race conditions may result. The first form is safe from within IRQ context; the second form is dangerous. However, the first form can lead to deadlock if it is used while a resource is being held that the handler may need; the second form may permit the resource to be freed.

It is important to notice that the enable/disable functions have a depth; if `disable_irq()` has been called twice, `enable_irq()` will have to be called twice before interrupts are handled again.

It is also possible to disable/enable all interrupts, in order to protect critical sections of code. This is best done with the appropriate **spinlock** functions:

```
unsigned long flags;
spinlock_t my_lock;
spinlock_init (&my_lock);
...
spin_lock_irqsave(&my_lock,flags);
...... critical code ........
spin_unlock_irqrestore(&my_lock,flags);
```

You should be very careful with the use of these functions as you can paralyze the system.

8.6 What You Cannot Do at Interrupt Time

Interrupts do not run in process context. Thus you cannot refer to `current` to access the fields of the `task_struct` as they are ill-defined at best. Usually `current` will point to whatever process was running when the interrupt service routine was entered, which has no *a priori* connection to the IRQ.

Anything which blocks can cause a kernel freeze, at least on the processor that blocks. In particular you cannot use any of the **sleep** functions, directly or indirectly. Indirect usage would happen for instance if you try to allocate memory with the flag `GFP_KERNEL` which can block if memory is not currently available, so you have to use `GFP_ATOMIC` instead, which returns on this situation.

You cannot call `schedule()` for similar reasons, or any call that indirectly calls the scheduler, such as all the **sleep** functions.

You cannot do a `down()` call on a **semaphore** as it can block while waiting for a resource. However, you can do an `up()` or any kind of `wake_up()` call.

You cannot request loading a module with `request_module()`.

You cannot transfer any data to or from a process's address space; i.e., no use of the `get_user()`, `put_user()`, `copy_to_user()`, `copy_from_user()` functions. These functions have the potential to go to sleep. Additionally, because there is no real user context, one can not transfer data to and from user-space using these functions.

8.7 IRQ Data Structures

The basic data structures involving IRQ's are defined in **/usr/src/linux/include/linux/irq.h** and **/usr/src/linux/include/linux/interrupt.h**.

For each IRQ there is a **descriptor** defined as:

```
2.6.31: 167 struct irq_desc {
2.6.31: 168     unsigned int            irq;
2.6.31: 169     struct timer_rand_state *timer_rand_state;
2.6.31: 170     unsigned int            *kstat_irqs;
2.6.31: 171 #ifdef CONFIG_INTR_REMAP
2.6.31: 172     struct irq_2_iommu      *irq_2_iommu;
2.6.31: 173 #endif
2.6.31: 174     irq_flow_handler_t      handle_irq;
2.6.31: 175     struct irq_chip         *chip;
2.6.31: 176     struct msi_desc         *msi_desc;
2.6.31: 177     void                    *handler_data;
2.6.31: 178     void                    *chip_data;
2.6.31: 179     struct irqaction        *action;        /* IRQ action list */
2.6.31: 180     unsigned int            status;         /* IRQ status */
2.6.31: 181
2.6.31: 182     unsigned int            depth;          /* nested irq disables */
2.6.31: 183     unsigned int            wake_depth;     /* nested wake enables */
2.6.31: 184     unsigned int            irq_count;      /* For detecting broken IRQs */
2.6.31: 185     unsigned long           last_unhandled; /* Aging timer for unhandled count */
2.6.31: 186     unsigned int            irqs_unhandled;
2.6.31: 187     spinlock_t              lock;
```



```
2.6.31: 188 #ifdef CONFIG_SMP
2.6.31: 189     cpumask_var_t           affinity;
2.6.31: 190     unsigned int            node;
2.6.31: 191 #ifdef CONFIG_GENERIC_PENDING_IRQ
2.6.31: 192     cpumask_var_t           pending_mask;
2.6.31: 193 #endif
2.6.31: 194 #endif
2.6.31: 195     atomic_t                threads_active;
2.6.31: 196     wait_queue_head_t       wait_for_threads;
2.6.31: 197 #ifdef CONFIG_PROC_FS
2.6.31: 198     struct proc_dir_entry   *dir;
2.6.31: 199 #endif
2.6.31: 200     const char              *name;
2.6.31: 201 } ____cacheline_internodealigned_in_smp;
```

`status` can be one of the following values:

Table 8.2: **IRQ status values**

Value	Meaning
IRQ_INPROGRESS	The handler for this IRQ handler is currently being executed.
IRQ_DISABLED	The IRQ line has been disabled.
IRQ_PENDING	An IRQ has occurred and been acknowledged, but not yet serviced.
IRQ_REPLAY	The IRQ line has been disabled but the previous occurrence on this line has not yet been acknowledged.
IRQ_AUTODETECT	The kernel is trying auto-detection on this IRQ line.
IRQ_WAITING	The kernel is trying auto-detection on this IRQ line and no interrupts have yet been detected.
IRQ_LEVEL	The IRQ line is level-triggered.
IRQ_MASKED	The IRQ line is masked and shouldn't be seen again.
IRQ_PER_CPU	The IRQ is per CPU.

`action` lists the service routines associated with the IRQ; the element points to the first `irqaction` structure in the list. We'll describe this structure in detail.

`depth` is 0 if the IRQ line is enabled. A positive value indicates how many times it has been disabled. Each `disable_irq()` increments the counter and each `enable_irq()` decrements it until it reaches 0 at which point it enables it. Thus this counter is used as a semaphore.

lock is used to prevent race conditions.

The `irqaction` structure looks like:

```
2.6.31:  93 struct irqaction {
2.6.31:  94     irq_handler_t handler;
2.6.31:  95     unsigned long flags;
2.6.31:  96     cpumask_t mask;
2.6.31:  97     const char *name;
2.6.31:  98     void *dev_id;
2.6.31:  99     struct irqaction *next;
2.6.31: 100     int irq;
2.6.31: 101     struct proc_dir_entry *dir;
2.6.31: 102     irq_handler_t thread_fn;
2.6.31: 103     struct task_struct *thread;
2.6.31: 104     unsigned long thread_flags;
2.6.31: 105 };
```

`handler` points to the interrupt service routine, or handler, that is triggered when the interrupt arrives. We'll discuss the arguments later.

`flags` is a mask of the following main values:

Table 8.3: **IRQ handler flags**

Flag	Meaning
IRQF_DISABLED	The interrupt runs with interrupts disabled; i.e., it is a **fast** handler.
IRQF_SHARED	The IRQ may be shared with other devices, if they all mutually agree to it.
IRQF_SAMPLE_RANDOM	The IRQ line may contribute to the **entropy pool** which the system uses to generate random numbers which are used for purposes like encryption. This should not be turned on for interrupts which arrive at predictable times.
IRQF_PROBE_SHARED	Set when sharing mismatches are expected to occur.
IRQF_TIMER	Set to indicate this is a timer interrupt handler.
IRQF_NOBALANCING	Set to exclude this interrupt from irq balancing.
IRQF_IRQNOPOLL	Interrupt is used for polling (only the interrupt that is registered first in an shared interrupt is considered for performance reasons)

Note that if the IRQ line is being shared, the `IRQF_DISABLED` flag will be effective only if it is specified on the first handler registered for that IRQ line.

`mask` indicates which interrupts are blocked while running.

`name` points to the identifier that will appear in **/proc/interrupts**.

`dev_id` points to a unique identifier in the address space of the device driver (or kernel subsystem) that has registered the IRQ. It is used as a **cookie** to distinguish among handlers for shared IRQ's and is important for making sure the right handler is deregistered when a request is made. Device drivers often have it point to a data structure which the handler routine will have access to. If the IRQ is not being shared, `NULL` can be used.

`next` points to the next `irqaction` structure in the chain that are sharing the same IRQ.

8.8 Installing an Interrupt Handler

Normally device drivers do not directly access the data structures we just described. Instead they use the following functions to install and uninstall interrupt handlers:

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

int request_irq (unsigned int irq,
    irqreturn_t (*handler)(int irq, void *dev_id);
    unsigned long flags,
    const char *device,
    void *dev_id);
void synchronize_irq (unsigned int irq);
void free_irq (unsigned int irq, void *dev_id);
```

`irq` is the interrupt number. It is used only if the handler can be used for more than one interrupt.

`handler()` is the handler to be installed.

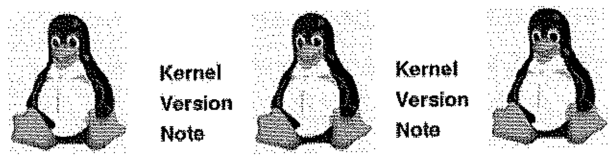
`flags` is the same bit-mask of options we described before; i.e., `IRQF_DISABLED` etc. Requesting sharing when the IRQ has been already registered as non-sharing may generate verbose but harmless debugging messages.

`device` is the same as the `name` field in the `irqaction` structure; it sets the identifier appearing in **/proc/interrupts**:

`dev_id` is the same unique identifier used for shared IRQ lines that appeared in the `irqaction` structure.

The `handler()` function has two arguments:

- `irq` is useful if more than one IRQ is being serviced.
- `dev_id` is used for shared interrupts.



- Kernels earlier than 2.6.19 contained a third argument, a data structure of type `pt_regs`, which holds a snapshot of the processor's context before the interrupt. It is used mostly for debugging, and for restoring the register state after the interrupt is handled. The precise definition of this structure is CPU-dependent; see **/usr/src/linux/arch/x86/include/asm/ptrace.h**.
- The `pt_regs` argument was removed in the 2.6.19 kernel, as it was rarely used in drivers, and eliminating it saved stack space and code, and boosted performance somewhat.
- If access to this structure should happen to be needed, the inline function `struct pt_regs *regs = get_irq_regs();` can be used.

`request_irq()` can be called either upon device initialization or when the device is first used (`open()`). Before the introduction of **udev** it often was better to do it in `open()` when the device was first used. However, with **udev** one doesn't tend to pre-load devices. This function should be called **before** the device is sent an instruction to enable generation of interrupts.

`free_irq()` can be called either during cleanup or `release()` (close). This function should be called only after the device is instructed not to interrupt the CPU anymore.

Before calling `free_irq()` one should call `synchronize_irq()` which ensures that all handlers for this particular **IRQ** are finished running before the free request is made. One should be careful that this function does not block.

The interrupt handler returns a value of type `irqreturn_t`. The three possible return values are:

Table 8.5: **IRQ handler return values**

Return Value	Meaning
`IRQ_NONE`	The handler didn't recognize the event; i.e., it was due to some other device sharing the interrupt, or it was spurious.
`IRQ_HANDLED`	The handler recognized the event and did whatever was required.
`IRQ_RETVAL(x)`	Evaluates as `IRQ_HANDLED` if the argument is non-zero; `IRQ_NONE` otherwise.

If no registered handler returns `IRQ_HANDLED` for a given IRQ, it is assumed to be spurious and a warning messaged is printed. Note that there is still no **consumption** of interrupts; all registered handlers are still called for a given IRQ line, even if one or more of them claims to have handled the event.

8.9 Labs

Lab 1: Shared Interrupts

Write a module that shares its IRQ with your network card. You can generate some network interrupts either by browsing or pinging. (If you have trouble with the network driver, try using the mouse interrupt.)

Check `/proc/interrupts` while it is loaded.

Have the module keep track of the number of times the interrupt handler gets called.

Lab 2: Sharing All Interrupts

Extend the previous solution to construct a character driver that shares every possible interrupt with already installed handlers.

The highest interrupt number you have to consider will depend on your kernel and platform; look at **/proc/interrupts** to ascertain what is necessary.

Take particular care when you call `free_irq()` as it is very easy to freeze your system if you are not careful.

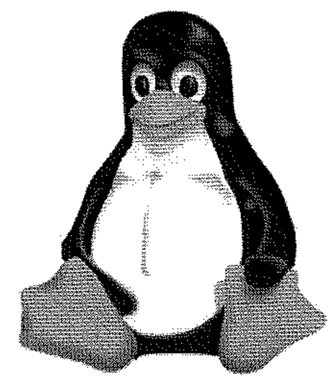
The character driver can be very simple; for instance if no `open()` and `release()` methods are specified, success is the default.

A `read()` on the device should return a brief report on the total number of interrupts handled for each **IRQ**.

To do this you'll also have to write a short application to retrieve and print out the data. (Don't forget to create the device node before you run the application.)

Chapter 9

Modules II: Exporting, Licensing and Dynamic Loading



We'll see how symbols are **exported** from the kernel to modules and from one module to another. We'll also discuss some aspects of **module licensing**. We'll consider demand and dynamic loading and unloading of modules. We'll see what changes are necessary in order to make a driver part of the kernel proper instead of a module, and discuss some details of the kernel and module build process.

9.1 Exporting Symbols

In order for built-in kernel code to make a symbol (i.e., a variable or function) available for use by modules, it has to properly **export** it. If a module has symbols which are to be used by modules which are loaded after it is, it also has to export the symbol.

This is accomplished with the use of the `EXPORT_SYMBOL()` macro:

```
int my_variable;
int my_export_fun (){ ....... };
EXPORT_SYMBOL(my_variable);
EXPORT_SYMBOL(my_export_fun);
```

Note that the symbols will be exported even if they are declared as `static`.

It is also possible to export symbols with the macro:

```
EXPORT_SYMBOL_GPL();
```

Exactly how this macro should be used and interpreted has sometimes been controversial. Certainly it means quite literally the symbol can be exported only to modules which are licensed under the **GPL**; e.g., it can't be used in binary-only drivers. However, some feel it should be done only for modules which are used internally by the kernel for basic functions.

There are some other specialized methods of exporting symbols:

```
EXPORT_PER_CPU_SYMBOL();
EXPORT_PER_CPU_SYMBOL_GPL();
EXPORT_SYMBOL_GPL_FUTURE();
EXPORT_UNUSED_SYMBOL();
EXPORT_UNUSED_SYMBOL_GPL();
```

Kernels earlier than the 2.6 series exported all global symbols in a module **unless** they were explicitly declared as `static`; that is one reason why you see the `static` keyword so liberally used in kernel code, for the purpose of avoiding *name pollution*.

Note that is still makes sense to declare symbols as `static`; if the code is compiled as built-in the symbols would be globally visible as the kernel is just one big program. Indeed the usual rule of thumb is that all symbols should be declared `static` unless there is a need to do otherwise.

9.2 Module Licensing

Modules can be **licensed** with the `MODULE_LICENSE()` macro, as in:

```
....
MODULE_DESCRIPTION("Does Everything");
MODULE_AUTHOR("Vandals with Handles");
MODULE_LICENSE("GPL v2");
....
```


Besides the informational content, this macro has important consequences: Any other license causes the entire kernel to be **tainted**, and warning messages appear when the module is loaded. For the most part, any system problems, crashes etc. that arise while using a tainted kernel are most likely to be ignored by kernel developers. (The pseudo-file **/proc/sys/kernel/tainted** shows your kernel's status.)

The following licenses are understood by the kernel:

Table 9.1: **Licenses**

License	Meaning	Tainted?
GPL	GNU Public License, V2 or later	No
GPL v2	GNU Public License, V2	No
GPL and additional rights	GNU Public License, V2 rights and more	No
Dual BSD/GPL	GNU Public License, V2 textbfor BSD license choice	No
Dual MPL/GPL	GNU Public License, V2 **or** Mozilla license choice	No
Dual MIT/GPL	GNU Public License, V2 **or** MIT license choice	No
Proprietary	Non free products (as in freedom, not free beer)	Yes

You can see the licenses of loaded modules with a script like:

```
#!/bin/bash
for names in $(cat /proc/modules | awk ' {print $1;} ')
        do echo -ne "$names\t     \t"
        modinfo $names | grep  license
done
```

- A phrase sometimes heard in the **Linux** kernel developer community is:
  All binary modules are illegal.
- A thorough debate on this topic was held on the kernel mailing list in late 2006. (See **http://lwn.net/Articles/215075** for a summary.) The main view coming out of that discussion stated that binary modules could **not** be banned, because the **GPL** controls **distribution** and not **use** of code.
- We don't want to get into a legal discussion here, but it seems clear that whether or not certain practices were accepted in the past, the future trend is that it is only going to become more difficult to get away with binary modules.
- Even if legal enforcement is not pursued vigorously, it is clear that increasing technical impediments and inefficiencies will make going proprietary more difficult and more expensive if not downright impossible.

9.3 Automatic Loading/Unloading of Modules

request_module():

A module may explicitly request loading of one or more other modules through:

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

int request_module (constant char *name, ...)
```

The module **name** will be dynamically loaded (using **modprobe**) together with any other required modules. **name** may be the actual name of the module or an alias specified in **/etc/modprobe.conf**. The current process will sleep until the module is loaded.

The additional, variable number of arguments to `request_module()` represent a formatted list in the manner of `printf()`. So for example one could do:

```
request_module("my-device-%d", device_number);
```

Dynamic loading can occur **only in the context of a process**; i.e., you will get errors if you call `request_module()` from an interrupt handler. It **must** be called from a driver entry point function.

The return value of `request_module()` is not very useful; success implies only that the request was properly executed, not that it succeeded. If you crawl through the source, you'll see an `exec()` of **modprobe** is requested and the error status is that reported for the `exec()`. Here the kernel actually makes an excursion to user-space.

Note that the requesting module can not call functions in the requested module; if that were so it would not be able to load in the first place (due to unresolved references.) In some sense, therefore, this function is a kind of *pre-fetch.*

Demand Loading:

A module may require other modules to be loaded in a stack. This may be accomplished either by:

- Loading them in the proper order with **insmod**.
- Loading them as a stack with **modprobe**. (Note that **depmod** must have been run previously, and the modules to be demand-loaded must be located in a place known to **depmod**).

Dynamic Loading:

Often one will want a module to be auto-loaded whenever its corresponding device node is accessed by an application. An example would be having the sound driver loaded every time `/dev/dsp` is read or written to by applications.

This can be accomplished by inserting lines in `/etc/modprobe.conf` of the form


```
alias block-major-254-* mybdrv
alias  char-major-254-* mycdrv
```

where we have used a wildcard for the minor number. (Note the 2.4 kernel only took a major number, and that form will still work.)

This assumes that `mybdrv.ko` and `mycdrv.ko` can be found in the path searched by **depmod** and that you have accessed the device nodes with this major number. Note that the name of the device node is invoked by the application and need not match the module name. You may also use such names as arguments to `request_module()`.

Note that the use of **udev** has reduced the need for this technique. However, the underlying methods are quite different. **udev** loads driver modules upon **discovery** of the device; using these aliases loads them upon first **use** of the device, which should come later, at least for hardware drivers.

9.4 Built-in Drivers

Device drivers and other facilities can be loaded either as an integral part of the kernel, or as modules, and many kernel components have the capability of being used either way. Inclusion in the kernel requires kernel re-compilation of the entire kernel; modularization does not.

At most only minor changes are necessary to the code; most often none are required. However, the kernel configuration files must deal with all three possibilities; built-in, module, or neither.

It is possible to mark both data and functions for removal after kernel initialization. This is done with the keywords `__init` and `__initdata`. So for example, if you have

```
static int some_data __initdata = 1 ;
void __init somefunc (void) { ... }
```

the data and code will go into a special initialization section of the kernel and be discarded after execution. One has to be careful that the code or data is not referenced after **init** starts. You may have noticed messages to this effect during the system boot:

```
[0.841689] Freeing unused kernel memory: 424k freed
```

The `__exit` macro doesn't do much except group all such labeled material together in the executable, in an area not likely to be cached.

If you use the `module_init()`, `module_exit()` macros, you should be able to avoid using any `#ifdef MODULE` statements in your code.

If `MODULE` is not defined, any function referenced by the `module_exit()` macro is dropped during compilation, since built-in drivers never get unloaded.

In addition, the kernel arranges for automatic loading of all `module_init()` functions, using the following recipe:

First, the `module_init()` macro in **/usr/src/linux/include/linux/module.h** will create a section in the `.o` file named `.initcall.init` This section will contain the address of the module's init function.

Thus, when all of the `.o` files are linked together, the final object file will contain a section called `.initcall.init` which becomes, in effect, an array of pointers to all of the init functions.

It is possible to assign priorities to initialization calls; the code which sets this up is in **/usr/src/linux /include/linux/init.h**:

```
#define pure_initcall(fn)           __define_initcall("0",fn,0)

#define core_initcall(fn)           __define_initcall("1",fn,1)
#define core_initcall_sync(fn)              __define_initcall("1s",fn,1s)
#define postcore_initcall(fn)               __define_initcall("2",fn,2)
#define postcore_initcall_sync(fn)  __define_initcall("2s",fn,2s)
#define arch_initcall(fn)           __define_initcall("3",fn,3)
#define arch_initcall_sync(fn)              __define_initcall("3s",fn,3s)
#define subsys_initcall(fn)         __define_initcall("4",fn,4)
#define subsys_initcall_sync(fn)    __define_initcall("4s",fn,4s)
#define fs_initcall(fn)                     __define_initcall("5",fn,5)
#define fs_initcall_sync(fn)                __define_initcall("5s",fn,5s)
#define rootfs_initcall(fn)         __define_initcall("rootfs",fn,rootfs)
#define device_initcall(fn)         __define_initcall("6",fn,6)
#define device_initcall_sync(fn)    __define_initcall("6s",fn,6s)
#define late_initcall(fn)           __define_initcall("7",fn,7)
#define late_initcall_sync(fn)              __define_initcall("7s",fn,7s)

#define __initcall(fn) device_initcall(fn)

#define __exitcall(fn) \
    static exitcall_t __exitcall_##fn __exit_call = fn

#define console_initcall(fn) \
    static initcall_t __initcall_##fn \
    __used __section(.con_initcall.init) = fn

#define security_initcall(fn) \
    static initcall_t __initcall_##fn \
    __used __section(.security_initcall.init) = fn
```

so that default is priority 6. (Note that in the 2.6.19 kernel, additional sublevels such as **6s** were introduced.)

The routine in **/usr/src/linux/init/main.c** that calls all the functions is:

```
static void __init do_initcalls(void)
{
    initcall_t *call;

    for (call = __early_initcall_end; call < __initcall_end; call++)
        do_one_initcall(*call);

    /* Make sure there is no pending stuff from the initcall sequence */
    flush_scheduled_work();
}
```

Note that in addition to using the `module_init()` macro, each driver still should use the `__init`

attribute when defining the body of the init function. This places the code in the section `.text.init`, which is the section that is reclaimed.

9.5 Kernel Building and Makefiles

The **Linux** kernel building process has become quite complex, and was completely reworked for the 2.6 kernel. Fortunately, using it is quite easy. Full documentation can be found under **/usr/src/linux /Documentation/kbuild.**

Important components include:

- The top-level `Makefile`.
- The configuration file, `.config`.
- The top-level architecture-dependent `Makefile`.
- Subdirectory `Makefiles`.
- In each directory with a `Makefile`, there is a file named `Kconfig`, which interfaces with the kernel configuration utilities.

The documentation that comes with the kernel does an excellent job of explaining the relationship of these quantities, so we won't try to repeat it.

Here is an example of a simple Makefile:

```
obj-$(CONFIG_FOO1)      += foo1.o
obj-$(CONFIG_FOO2)      += foo2.o
obj-$(CONFIG_FOO3)      += foo3.o

foo3-objs               := foo3a.o foo3b.o foo3c.o

EXTRA_CFLAGS            += -DFOO_DEBUG
```

(Note we have `.o`, not `.ko`.)

As the **make** proceeds, three environmental variables are constructed according to the `CONFIG_*` values:

- `obj-y`: Those source files to be compiled into the kernel itself.
- `obj-m`: Those source files to be compiled into modules.
- `obj-`: Those source files to be ignored.

If more than one file must be compiled and linked together that is done as in the `foo3-objs` example.

The variable `EXTRA_CFLAGS` can be used to augment compiler flags.

To get your new facility in the configuration utilities requires modifying one more file, `Kconfig` in the same directory, which is written in a customized scripting language, but is easy to hack. This consists of a series of sections such as:

```
config FOO1
        bool "FOO1 Driver"
        default y
        help
          Here is the help item on the foo1 driver.
config FOO2
        tristate "FOO2 Driver"
        default n
        help
          Here is the help item on the foo2 driver.
```

There are other directives in this file for indicating dependencies, etc.

Slightly fancier **Makefiles** are required if you need to recurse through subdirectories, etc, but looking at the examples in the kernel source gives good enough guidance.

To repeat, there are three main ingredients; the `Makefile`, the `Kconfig` file, and the source itself.

9.6 Labs

Lab 1: Stacked Modules

Write a pair of modules one of which uses a function defined in the other module.

Try loading and unloading them, using **insmod** and **modprobe**.

Lab 2 Duplicate Symbols

Copy your first module to another file, compile and try to load both at the same time:

```
$ cp lab1_module1.c lab1_module1A.c
.... modify Makefile and compile
$ insmod lab1_module1.ko
$ insmod lab1_module1a.ko
```

Does this succeed?

Install your modules with `make modules_install`.

See how **depmod** handles this by. analyzing the `modules.dep` file that results.

Lab 3: Dynamic Module Loading

Take your basic character driver from the previous exercise and adapt it to use dynamic loading:

Construct a trivial second module and have it dynamically loaded during the character driver's `open()` entry point. (Make sure the name of the file that is requested is the same as the name of your file.)

Add a small function to your character driver and have it referenced by the second module.

Make sure you place your modules in a place where **modprobe** can find them, (Installing with the target `modules_install` will take care of this for you.)

You can use either **cat** or the main program from the character driver lab to exercise your module. What happens if you try to request loading more than once?

Lab 4: Demand Loading of Drivers

Make your character driver load upon use of the device; i.e., when you do something like

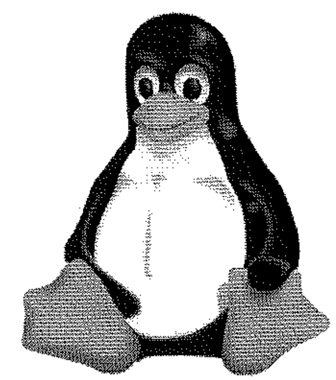
```
cat file > /dev/mycdrv
```

have the driver load.

Make the adjustments to `/etc/modprobe.conf` as needed and put the module in the proper place with `make modules_install`.

Chapter 10

Debugging Techniques



We'll consider various techniques used to debug device drivers and the kernel. We'll discuss dissecting **oops** messages, and direct use of debuggers, focusing on the **kdb** tool, and including **kprobes**. We'll also consider the use of **debugfs**.

10.1 oops Messages

oops messages indicate that a fault occurred in kernel mode. Depending on the nature of the fault that produced the **oops**, the fault may be fatal, serious, or inconsequential.

If the oops occurs in process context the kernel will attempt to back out of the current task, probably killing it. If it occurs in interrupt context the kernel can't do this and will crash, as it will if it occurs in either the idle task (pid=0) or init (pid=1).

The information provided contains a dump of the processor registers at the time of the crash and and a call trace indicating where it failed. Sometimes this is all one may need.

Getting the most use out of **oops** messages, and almost all kernel debugging techniques, requires having at least some familiarity with assembly language. For an example of how to work through an **oops** message see **http://lkml.org/lkml/2008/1/7/406**.

In order to cause an **oops** deliberately, one can do

```
if (disgusting_condition)
    BUG();
```

or

```
BUG_ON(disgusting_condition);
```

One can also induce a system crash while printing out a message such as:

```
if (fatal_condition)
   panic ("I'm giving up because of task %d\n", current->pid);
```

- The website **http://www.kerneloops.org** maintains a database of current **oops** and has helped kernel developers debug successfully.

Here is a trivial module (`crashit.c`) that contains a null pointer dereference that can trigger an **oops** message:

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

static int __init my_init (void)
{
     int *i;
     i = 0;
     printk (KERN_INFO "Hello: init_module loaded at address 0x%p\n",
             init_module);
     printk (KERN_INFO "i=%d\n", *i);
     return 0;
}

static void __exit my_exit (void)
{
     printk (KERN_INFO "Hello: cleanup_module loaded at address 0x%p\n",
             cleanup_module);
}
```



```
module_init (my_init);
module_exit (my_exit);

MODULE_LICENSE ("GPL v2");
```

We can disassemble the code with **objdump**. Doing

```
objdump -d crashit.ko
```

gives

```
crashit.ko:     file format elf32-i386

Disassembly of section .init.text:

00000000 <init_module>:
   0:       83 ec 08                     sub    $0x8,%esp
   3:       c7 44 24 04 00 00 00         movl   $0x0,0x4(%esp)
   a:       00
   b:       c7 04 24 00 00 00 00         movl   $0x0,(%esp)
  12:       e8 fc ff ff ff               call   13 <init_module+0x13>
  17:       a1 00 00 00 00               mov    0x0,%eax
  1c:       c7 04 24 00 00 00 00         movl   $0x0,(%esp)
  23:       89 44 24 04                  mov    %eax,0x4(%esp)
  27:       e8 fc ff ff ff               call   28 <init_module+0x28>
  2c:       31 c0                        xor    %eax,%eax
  2e:       83 c4 08                     add    $0x8,%esp
  31:       c3                           ret
Disassembly of section .exit.text:

00000000 <cleanup_module>:
   0:       83 ec 08                     sub    $0x8,%esp
   3:       c7 44 24 04 00 00 00         movl   $0x0,0x4(%esp)
   a:       00
   b:       c7 04 24 2c 00 00 00         movl   $0x2c,(%esp)
  12:       e8 fc ff ff ff               call   13 <cleanup_module+0x13>
  17:       83 c4 08                     add    $0x8,%esp
  1a:       c3                           ret
```

We produce the **oops** by attempting to load `crashit.ko`; it hangs during the initialization step, and produces the following **oops** message (which gets appended to `/var/log/messages`):

```
Hello: init_module loaded at address 0xf8a07000
Unable to handle kernel NULL pointer dereference at virtual address 00000000
 printing eip:
f8a07017
*pde = 00000000
Oops: 0000 [#1]
PREEMPT
Modules linked in: crashit w83627hf eeprom lm75 i2c_sensor i2c_isa
 i2c_ viapro i2c_dev i2c_core sunrpc binfmt_misc uhci_hcd ehci_hcd
```

```
 snd_via82xx snd_ac97_codec snd_pcm_oss snd_mixer_oss snd_pcm snd_timer
 snd_page_alloc snd_mpu401_uart snd_rawmidi snd_seq_device snd soundcore
 floppy sata_promise sata_via libata
CPU:    0
EIP:    0060:[<f8a07017>]    Not tainted VLI
EFLAGS: 00210296   (2.6.13)
EIP is at my_init+0x17/0x32 [crashit]
eax: 0000003f   ebx: f89f9300   ecx: c045a2d8   edx: 00000001
esi: e3aa7000   edi: c0000000   ebp: e3aa7000   esp: e3aa7f94
 ds: 007b   es: 007b   ss: 0068
Process insmod (pid: 4628, threadinfo=e3aa7000 task=eaf36bb0)
Stack: f89f901c f8a07000 c0154733 c051edc4 00000001 f89f9300 0804a018 00000000
       080489f4 c0103f5b 0804a018 00000aa1 0804a008 00000000 080489f4 bfc89888
       00000080 0000007b 0000007b 00000080 ffffe410 00000073 00200246 bfc89820
Call Trace:
 [<f8a07000>] my_init+0x0/0x32 [crashit]
 [<c0154733>] sys_init_module+0x193/0x290
 [<c0103f5b>] sysenter_past_esp+0x54/0x75
Code:  Bad EIP value.
```

It contains a dump of the processor registers at the time of the crash and and a call trace indicating where it failed. We see it failed in `crashit` at an offset of `0x17` bytes into `my_init()`. Comparing with the object dis-assembly tells us precisely what the offending line is.

10.2 Kernel Debuggers

The first thing to understand about using debuggers on the **Linux** kernel is that Linus Torvalds **hates** them For his entertaining explanation of why, see **http://lwn.net/2000/0914/a/lt-debugger.php3**. The short explanation for this attitude is that reliance on debuggers can encourage fixing problems with band-aids rather than brains, and leads to rotten code.

Linus **will** tolerate (and even encourage) optional debugging aids that either check specific known errors (e.g., am I sleeping while holding a spinlock?), or require only an entry point into the kernel, and permit their work to be done through modules.

gdb

The **gdb** debugger can be used to debug a running kernel. The execution line would be:

```
$ gdb /boot/vmlinux /proc/kcore
```

where the first argument is the currently running *uncompressed* kernel. One can use **ddd** or another other graphical interfaces to **gdb**. The sections of the kernel being debugged must have been compiled with the `-g` option, to get much useful information. This is not for the faint-hearted. It is pretty difficult.

kdb

kdb is an interactive kernel debugger. It can be downloaded from **http://oss.sgi.com/projects/kdb**, and is furnished as one or more patches to the kernel, which include extensive documentation. It has the ability to:

- Examine kernel memory and data structures.
- Control operations, such as single-stepping, setting breakpoints.
- Get stack tracebacks, do instruction dis-assembly., etc.
- Switch CPUs in an SMP system.
- Etc.

kdb is automatically entered upon encountering an oops, a data access fault in kernel mode, using a **kdb** flag on the kernel command line, or using the **pause** key.

An informative tutorial on **kdb** has been published by **IBM DeveloperWorks**; it can be found at **http://www.ibm.com/developerworks/linux/library/l-kdbug/**.

kgdb

kgdb is another interactive kernel debugger. It originally required two computers to be connected through a null-modem serial cable. On the remote host system the user runs **gdb** (or a GUI wrapper to it such as **ddd**) and can then break into the kernel on the target system, setting breakpoints, examining data, etc. It is possible to stop the target machine kernel during the boot process.

kgdb was incorporated in the mainline kernel with kernel version 2.6.26. The oft-requested ability to debug through a network connection exists in the development version, but is as yet not working fully and thus is not included in the mainline version.

crash

The **crash** utility is probably provided by your **Linux** distribution and full documentation can be found at **http://people.redhat.com/anderson/**.

With **crash** one can examine all critical data structures in the kernel, do source code disassembly, walk through linked lists, examine and set memory, etc. **crash** can also examine **kernel core dump** files created by the **kdump, diskdump** and **xendump** packages.

- To use **crash** or **gdb** you need a kernel which has been compiled with `CONFIG_DEBUG_INFO=y`, and you need a copy of the *uncompressed* kernel, **vmlinux**.
- If you have compiled the kernel yourself this is no problem and you will find the uncompressed kernel in the main kernel source directory.
- However, if you are debugging a distributor-provided kernel, it will probably require a little more work. On **RPM**-based systems you will also need to install the distributor's **kernel-debuginfo** package, which contains all the symbols and information stripped from the widely distributed kernel.

10.3 debugfs

The **debugfs** filesystem appeared in the 2.6.11 kernel. It can be used as a simpler and more modern alternative to using the **/proc** filesystem, which has an inconvenient interface and which kernel developers have lost their taste for.

The main purpose of **debugfs** is for easy access to debugging information, and perhaps to set debugging behaviour. It is meant to be accessed like any other filesystem, which means standard reading and writing tools can be used.

One can also use **sysfs** for the same purposes. However, **sysfs** is intended for information used in system administration, and is also meant to be based in a coherent way on the system's device tree as mapped out along the system buses.

The code for **debugfs** was developed mainly by Greg Kroah-Hartman. The functions used are:

```
#include <linux/fs.h>
#include <linux/debugfs.h>

struct dentry *debugfs_create_dir (const char *name, struct dentry *parent)
struct dentry *debugfs_create_file (const char *name, mode_t mode, struct dentry *parent,
                                    void *data, struct file_operations *fops);
void debugfs_remove (struct dentry *dentry);
```

As with **/proc** you can create your own entry under the **debugfs** root directory by creating a directory with `debugfs_create_dir()`; supplying `NULL` for `parent` in the above functions places entries in the root directory. The `mode` argument is the usual filesystem permissions mask, and `data` is an optional parameter that can be used to point to a private data structure.

The `fops` argument point to a `file_operations` structure containing a jump table of operations on the entry, just as it is used in character drivers. One probably needs to supply only the ownership field, and reading and writing entry point functions.

For the read function one may want to take advantage of the function:


```
ssize_t simple_read_from_buffer (void __user *to, size_t count, loff_t *ppos,
                                 const void *from, size_t available);
```

which is a convenience function for getting information from the kernel buffer pointed to by `from` into the user buffer `to` (using `copy_to_read()` properly), where the position `ppos` is advanced no further than `available` bytes. An example of a read function using this:

```
static ssize_t
my_read (struct file *file, char *buf, size_t count, loff_t * ppos)
{
    int nbytes;
    nbytes=sprintf(kstring, "%d\n", val);
    return simple_read_from_buffer (buf, count, ppos, kstring, nbytes);
}
```

Even simpler is to use are the convenience functions:

```
struct dentry *debugfs_create_u8   (const char *name, mode_t mode,
                                    struct dentry *parent, u8 *val);
struct dentry *debugfs_create_u16  (const char *name, mode_t mode,
                                    struct dentry *parent, u16 *val);
struct dentry *debugfs_create_u32  (const char *name, mode_t mode,
                                    struct dentry *parent, u32 *val);
struct dentry *debugfs_create_bool (const char *name, mode_t mode,
                                    struct dentry *parent, u32 *val);
```

These create an entry denoted by `name`, under the parent directory, which is used to simply read in and out a variable of the proper type. Note the variable is sent back and forth as a string. Thus one can with simply one line of code (two including the header file!) create an entry!

In order to use the **debugfs** facility, it has to be compiled into the kernel, and mounted:

```
mount -t debugfs none /sys/kernel/debug
```

where any mount point can be selected, but the directory at `/sys/kernel/debug` has been created for this purpose if you wish to use it.

Regardless of how you create your entries they **must** be removed with `debugfs_remove()` on the way out, because, as usual, the kernel does no garbage collection.

For a recent review of **debugfs** and how to use it see **http://lwn.net/Articles/334546**.

10.4 kprobes and jprobes

The **kprobes** debugging facility (originally contributed by developers at **IBM**) lets you insert breakpoints into a running kernel at any known address. One can examine as well as modify processor registers, data structures, etc.

Up to four handlers can be installed:

- The **pre-handler** is called just before the probed instruction is executed.
- The **post-handler** is called just after the probed instruction is executed, if no exception is generated.
- The **fault-handler** is called whenever an exception is generated by the probed instruction.
- The **break-handler** is called whenever the probed instruction is being single stepped or break-pointed.

The basic functions and data structures are defined in **/usr/src/linux/include/linux/kprobes.h** and **/usr/src/linux/kernel/kprobes.c**:

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

int register_kprobe(struct kprobe *p);
void unregister_kprobe(struct kprobe *p);

struct kprobe {
   struct hlist_node hlist;

   /* location of the probe point */
   kprobe_opcode_t *addr;

  /* Allow user to indicate symbol name of the probe point */
     char *symbol_name;

  /* Offset into the symbol */
     unsigned int offset;

   /* Called before addr is executed. */
   kprobe_pre_handler_t pre_handler;

   /* Called after addr is executed, unless... */
   kprobe_post_handler_t post_handler;

   /* ... called if executing addr causes a fault (eg. page fault).
    * Return 1 if it handled fault, otherwise kernel will see it. */
   kprobe_fault_handler_t fault_handler;

   /* ... called if breakpoint trap occurs in probe handler.
    * Return 1 if it handled break, otherwise kernel will see it. */
   kprobe_break_handler_t break_handler;

   /* Saved opcode (which has been replaced with breakpoint) */
   kprobe_opcode_t opcode;

   /* copy of the original instruction */
   struct arch_specific_insn ainsn;
};

typedef int (*kprobe_pre_handler_t) (struct kprobe *, struct pt_regs *);
typedef int (*kprobe_break_handler_t) (struct kprobe *, struct pt_regs *);
typedef void (*kprobe_post_handler_t) (struct kprobe *, struct pt_regs *, unsigned long flags);
typedef int (*kprobe_fault_handler_t) (struct kprobe *, struct pt_regs *, int trapnr);
```


Note that the handler functions receive a pointer to a data structure of type `pt_regs`, which contains the contents of the processor registers. This is obviously architecture-dependent, and is detailed in **/usr/src/linux/arch/x86/include/asm/ptrace.h**.

The `flags` argument to the post handler is currently unused, and the `trapnr` argument to the fault handler gives which exception caused the fault.

In order to use **kprobes**, one must:

- Fill in the `kprobe` data structure with pointers to supplied handler functions.
- Supply either the address (`addr`) or symbolic name (`symbol_name` with an optional `offset`) where the probe is to be inserted.
- Call `register_kprobe()`, with a return value of `0` indicating successful probe insertion.

When finished one uses `unregister_kprobe()`, with an obvious catastrophe being the result if one forgets to do so.

The only remaining ingredient is to obtain the address of the probed instruction. If the symbol is exported, then you can merely point directly to it, as in

```
kp.addr = (kprobe_opcode_t *) mod_timer;
```

Even if the symbol is not exported you can still specify the name directly with something as simple as

```
kp.symbol_name = "do_fork";
```

One should not set **both** the address and the symbol, as that will lead to an error.

The additional **jprobe** facility lets you easily instrument any function in the kernel. The relevant registration and unregistration functions, and the new relevant data structure are:

```
int register_jprobe (struct jprobe *jp);
void unregister_jprobe (struct jprobe *jp);
void jprobe_return (void);

struct jprobe {
    struct kprobe kp;
    kprobe_opcode_t *entry;
}
```

In order to use this you have to set up a structure of type `jprobe`, in which the `entry` field points to a function of the exact same prototype and arguments as the function being probed, which should be pointed in the `kp.addr` field just as for **kprobes**.

The instrumentation function will be called every time the probed function is called and must exit with the function `jprobe_return()`. It is called **before** the probed function.

The contents of registers and the stack are restored before the function exits. However, changing the values of arguments can make a (possibly destructive) difference.

- One can turn **kprobes** on and off dynamically, even while it is currently in use. To do this you must mount the **debugfs** pseudo-filesystem:

```
$ mount -t debugfs none /sys/kernel/debug

$ ls -l /sys/kernel/debug/kprobes
total 0
-rw------- 1 root root 0 Jun 11 08:28 enabled
-r--r--r-- 1 root root 0 Jun 11 01:44 list
```

- By echoing 1 or 0 to `enabled` you can turn **kprobes** on and off. By looking at `list` you can examine all currently loaded probes.

SystemTap

While **kprobes** is very powerful, its use requires a relatively low level kernel incursion. **SystemTap** provides an infrastructure built on top of **kprobes** that simplifies writing, compiling and installing kernel modules, and gathering up useful output. The **SystemTap** project can be found at **http://sourceware.org/systemtap/**.

10.5 Labs

Lab 1: Using kprobes

Place a **kprobe** at an often executed place in the kernel. A good choice would be the `do_fork()` function, which is executed whenever a child process is born.

Put in simple handler functions.

Test the module by loading it and running simple commands which cause the probed instruction to execute, such as starting a new shell with **bash**.

Lab 2: Using jprobes

Test the **jprobes** facility by instrumenting a commonly used kernel function.

Keep a counter of how many times the function is called. If you print it out each time, be careful not to get overwhelmed with output.

Lab 3: Probing a module

Take an earlier module (such as a character driver) and add both **kprobes** and **jprobes** instrumentation to it.

Does the function you are probing need to be exported to be accessible to the probe utilities?

Lab 4: Using debugfs.

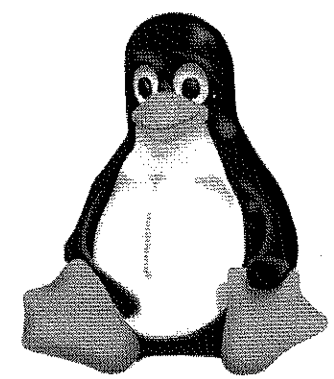
Write a module that creates entries in **debugfs**.

First use one of the convenience functions to make just a simple one variable entry under the root **debugfs** filesystem, of whatever length you desire.

Next create your own directory and put one or more entries in it.

Chapter 11

Timing and Timers



We'll consider the various methods **Linux** uses to manage time. We'll see how **jiffies** are defined and used, and how delays and timing are implemented. We'll discuss **kernel timers**, showing how they are used and how they are implemented in the **Linux** kernel. We'll also discuss the new **hrtimers** feature and its high resolution implementation.

11.1 Jiffies

A coarse time measurement is given by the variable `unsigned long volatile jiffies` defined in **/usr/src/linux/include/linux/jiffies.h**

Before kernel 2.6.21 `jiffies` was simply a counter that is incremented with every timer interrupt. However, with the the incorporation of `tickless` kernels one need not keep processing timer interrupts when the system is idle. (For full details, see **http://www.lesswatts.org/projects/tickless**.)

The default frequency is `HZ = 1000` on the **x86**, but is configurable at compile time, within a range of `HZ=100` to `HZ=1000`. Thus we obtain a resolution between 10 and 1 milliseconds for the `jiffies` value.

With `HZ=1000` `jiffies` will overflow (and wrap) at about 50 days of uptime; if someone has been sloppy, what will happen then is unpredictable. However, if you are writing device drivers it is unlikely you will reach that long an uptime.

To help avoid any potential problems, the `jiffies` value is set during boot to `INITIAL_JIFFIES = -300 HZ`, which causes the value to wrap after five minutes. As a side effect you may notice that the value of `jiffies` differs from the number of timer interrupts read from `/proc/interrupts` by the same value. (Tickless kernels also break this equality.)

Useful macros to compare relative `jiffies` values are:

```
time_after(a,b)
time_before(a,b)
time_after_eq(a,b)
time_before_eq(a,b)
```

where the first one is true if time `a` is after time `b`, and the second one is the inverse macro. The other two macros also check for equality.

Note that there exists a variable named `jiffies_64`. On 64-bit platforms this is the same as `jiffies`; on 32-bit platforms `jiffies` points to its lower 32 bits. Since `jiffies_64` won't wrap for almost 600 million years (with `HZ=1000`), one need not worry about it doing so.

One has to be careful when using the 64-bit counter (on 32-bit platforms) as access to the value is not atomic; to do so one needs to use

```
u64 get_jiffies_64(void);
```

to read the value. (Note you never set a value of course.)

A number of macros are provided to convert `jiffies` back and forth to other ways of specifying time:

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

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

unsigned long timeval_to_jiffies (struct timeval *val);
void jiffies_to_timeval (unsigned long jiffies, struct timeval *val);

unsigned int  jiffies_to_msecs (const unsigned long j);
unsigned int  jiffies_to_usecs (const unsigned long j);
unsigned long msecs_to_jiffies (const unsigned int m);
unsigned long usecs_to_jiffies (const unsigned int u);
```

where the `timeval` and `timespec` structures should be familiar from user-space:


```
struct timeval {
  long    tv_sec;  /* seconds */
  long    tv_usec; /* microseconds */
};
struct timespec {
  long    tv_sec;  /* seconds */
  long    tv_nsec; /* nanoseconds */
};
```

11.2 Time Stamp Counter

For Pentium or better, **x86** cpus include a 64-bit register called the time stamp counter (**tsc**). At every clock signal the register is incremented; i.e., a 1 GHZ cpu would increase the register every nanosecond.

In principle, the **tsc** can be used for high precision time measurements (in fact no higher precision can be obtained than the clock tick) but in order to convert to times the kernel has to be able to determine the clock signal frequency. This calibration can not be built into the kernel during compilation since a kernel may be compiled on one system and run on another. Furthermore, on modern CPU's the clock frequency can vary continuously during operation.

The macros which obtain the value of the time stamp counter are:

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

rdtsc (low,high);
rdtscl (low);
rdtscll (val);
```

The `rdtsc()` macro reads the full 64-bit value of the **TSC** and stuffs it into two 32 bit arguments. The `rdtscl()` macro gets only the lower 32 bits. This is usually sufficient (a 2 GHZ system would overflow in about 2 seconds.) A third macro, `rdtscll()`, gets the full 64-bit value and stuffs it in a **long long**, which is 64-bit on both 32- and 64-bit platforms.

Internally the kernel keeps the variable `unsigned long cpu_khz`, which gives the speed in kilohertz. You can find out your system's clock speed by reading **/proc/cpuinfo**. Note that if you have a variable speed CPU, such as on a laptop, the speed may vary according to power state.

Note that on **i386** systems, the `msr.h` header file containing the macros does not appear under `/usr/include`. (It does, however, for the **x86_64** platform.) To compile user-space applications, you'll either have to point to the kernel headers on the compile line (with `-I/lib/modules/$(uname -r)/build/include`), or explicitly include the macros:

```
#define rdtsc(low,high)    __asm__ __volatile__("rdtsc" : "=a" (low), "=d" (high))
#define rdtscl(low)        __asm__ __volatile__("rdtsc" : "=a" (low) : : "edx")
#define rdtscll(val)       __asm__ __volatile__("rdtsc" : "=A" (val))
```

These macros can also be used from user-space; make sure you compile with optimization on.

The **TSC** is subject to drift and coordination problems in multi-CPU systems and better clock sources exist. In particular recent CPU's include a **HPET** (**H**igh **P**recision **E**vent **T**imer) capable of very high precision, up to the nanosecond level. Thus device drivers should avoid using the TSC directly for any kind of timing measurements.

To get the current time with roughly microsecond resolution, use

```
#include <linux/time.h>
void do_gettimeofday (struct timeval *tv);
struct timeval {
   time_t      tv_sec;  /* seconds */
   suseconds_t tv_usec; /* microseconds */
};
```

using the `timeval` data structure that should be familiar from the `select()` function in user-space. The time returned by this function is measured in seconds since the **Epoch**, midnight on January 1, 1970.

11.3 Inserting Delays

jiffies can be used to introduce busy waiting; i.e.,

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

jifdone = jiffies + delay * HZ;
while ( time_before(jiffies,jifdone) )
  {
    /* do nothing */
  }
```

This is an idiotic thing to do; because `jiffies` is volatile, it is reread every time it is accessed. Thus this loop locks the CPU during the delay (except that interrupts may be serviced.)

For short delays, one can use the following functions:

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

void ndelay(unsigned long nanoseconds);
void udelay(unsigned long microseconds);
void mdelay(unsigned long milliseconds);
```

One should not expect true nanosecond resolution for `ndelay()`; depending on the hardware it will probably be closer to microseconds.

Another delaying method which does not involve busy waiting is to use the functions:

```
void msleep (unsigned int milliseconds);
unsigned long msleep_interruptible (unsigned int milliseconds);
```

If `msleep_interruptible()` returns before the sleep has finished (because of a signal) it returns the number of milliseconds left in the requested sleep period.

11.4 What are Dynamic Timers?

Dynamic timers (also known as **kernel timers**) are used to delay a function's execution until a specified time interval has elapsed. The function will be run on the CPU on which it is submitted.

Because a CPU may not be immediately available when it is time to execute the function, you are guaranteed only that the function will not run before the timer expires; practically speaking this means it should occur at most a clock tick afterwards, unless some greedy high latency task has been suspending interrupts.

While an explicit periodic scheduling function does not exist, it is trivial to make a timer function re-install itself recursively.

The function will not be run in a process context; it will run as a **softirq** in an atomic context. Thus one cannot do anything which can not be done at **interrupt time**; i.e., no transfer of data back and forth with user-space, no memory allocation with `GFP_KERNEL`, no use of semaphores, etc., as these methods can go to sleep.

11.5 Timer Functions

The important data structure and functions used by kernel timers are:

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

struct timer_list {
    struct list_head entry;
    unsigned long expires;
    void (*function)(unsigned long);
    unsigned long data;
    struct tvec_t_base_s *base;
};

void init_timer (struct timer_list *timer);
void add_timer  (struct timer_list *timer);
void mod_timer  (struct timer_list *timer, unsigned long expires);
int  del_timer  (struct timer_list *timer);
int  del_timer_sync (struct timer_list *timer);
```

`list` points to the doubly-linked circular list of kernel timers.

`expires` in measured in `jiffies`. It is an absolute value, not a relative one.

The function to be run is passed as `function()` and data can be passed to it through the pointer argument `data`.

`init_timer()` zeroes the previous and next pointers in the linked list.

`add_timer()` inserts the timer into the global timer list.

`mod_timer()` can be used to reset the time at which a timer expires.

`del_timer()` can remove a timer before it expires. It returns `1` if it deletes the timer, or `0` if it's too late because the timer function has already started executing. It is not necessary to call `del_timer()`

if the timer expires on its own.

`del_timer_sync()` makes sure that upon return the timer function is not running on any CPU. It helps avoid race conditions and is preferable to use on **SMP** systems.

A timer can reinstall itself to set up a periodic timer. This can be done in either of two ways:

```
....
init_timer(&t);
t.expires= jiffies + delay;
add_timer(&t);
```

or

```
....
mod_timer(&t, jiffies+delay);
```

which is often done as a more compact form. Note that it is very important to reinitialize the timer when reinstalling; `mod_timer()` does this under the hood.

Example:

```
static struct timer_list my_timer;

init_timer(&my_timer);
my_timer.function = my_function;
my_timer.expires = jiffies+ticks;
my_timer.data = &my_data;

add_timer(&my_timer);
....
del_timer(&my_timer);
....
void my_function(unsigned long var){ };
```

11.6 Timer Implementation

The implementation of dynamic timers has to take care to two distinct tasks:

- Functions have to be inserted in and removed from the list of timers, or have the expiration times modified.
- Functions have to be executed at the proper time.

The simplest implementation would be to maintain one linked list of kernel timers, and add the newly requested timer function into the list either at the tail, or in some kind of sorted fashion, and then when the kernel decides to run any scheduled timer functions, scan the list and run those whose time value has expired.

However, this would be hideously inefficient. There may be many, even thousands, of functions whose expiration times might need to be scanned, and the kernel would be strangled by this task. Sorting might help the scanning process, but it would be paid for by expensive insertion and deletion operations.

Linux has implemented a very clever method in which it actually maintains 512 doubly-linked circular lists, so that the `next` and `prev` fields in the `timer_list` struct point only within one of the lists at any given time. Which list depends on the value of the `expires` field.

These lists are further partitioned into 5 groups:

Table 11.1: **Timer groups**

Group	Ticks	Time (for HZ=1000)
tv1	< 256	$< .256$ secs
tv2	$< 2^{14}$	< 16.4 secs
tv3	$< 2^{20}$	< 17.5 mins
tv4	$< 2^{26}$	18.6 hrs
tv5	$< \infty$	$< \infty$

The **tv1** group has a vector of 256 doubly-linked lists, set up so those in the first list will expire in the next timer tick, those in the second group will expire on the tick after that, and so on. Likewise, the **tv2-5** groups each have a vector of 64 doubly-linked lists, ordered in time groups.

Each time there is a timer tick, an index into the **tv1** list of vectors is incremented by one, and the timer functions which need to be launched are all in one doubly-linked list. When this index reaches 256, the function `cascade_timers()` gets called, which brings the first group of **tv2** in to replenish **tv1**, the first group of **tv3** in to replenish **tv2**, etc.

11.7 High Resolution Timers

The **Linux** kernel approach to dynamic timer implementation, while being quite clever and efficient, received a great enhancement with the addition of a new approach, gradually introduced since the 2.6.16 kernel

We begin with the observation that there are really two kind of dynamic timers:

- **Timeout** functions, found primarily in networking code and device drivers, used to signal when an event does not happen within a specified window of time, and either a task should be dropped or a recovery action initiated.
- **Timer** functions, expected to actually run within a specified latency and sequence.

Timeout functions tend to be far more numerous than timer ones, and thus in the present dynamic timer implementation, removal of a timer before it runs is far more frequent than actually running the function.

Timeout functions generally have only a weak precision requirement, while (in principle at least) timer functions may have more stringent needs.

The original implementation works very well for timeout functions, particularly because it does such a rapid job of timer removal, since only an index look up is required, and jiffies-level accuracy is generally fine.

The **hrtimers** (High Resolution Timers) **API**, introduced in the 2.6.16 kernel, is designed for dynamic timers actually expected to execute.

Rather than having a complex set of lists, there is only one list (per CPU) sorted by time of expiration, using a red-black tree algorithm. While insertion and removal may be somewhat slower than in the original method, there will be no need for a cascade operation and there are fewer elements in the list.

Expiration periods for the **hrtimers** are be expressed in nanoseconds rather than jiffies, although some platforms may still operate at lower resolution.

11.8 Using High Resolution Timers

The **hrtimers** feature required introduction of a new (and opaque) ktime_t which measures time with nanosecond resolution if the particular architecture can support it. The internal representation of ktime_t is quite different on 64-bit and 32-bit platforms and should not be monkeyed with.

Functions for dealing with this new time variable are contained in **/usr/src/linux/include/linux/ktime.h** and don't require much explanation:

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

ktime_t ktime_set (const long secs, const unsigned long nsecs);
ktime_t ktime_add (const ktime_t kt1, const ktime_t kt2);
ktime_t ktime_sub (const ktime_t kt1, const ktime_t kt2);
ktime_t ktime_add_ns (const ktime_t kt, u64 ns);
ktime_t ktime_get (void) /* monotonic time */
ktime_t ktime_get_real (void) /* real (wall) time */

ktime_t timespec_to_ktime (const struct timespec tspec);
ktime_t timeval_to_ktime (const struct timeval tval);
struct timespec ktime_to_timespec (const ktime_t kt);
struct timeval ktime_to_timeval (const ktime_t kt);
u64 ktime_to_ns (const ktime_t kt);
```

The high resolution timers are controlled with the functions defined in **/usr/src/linux/include/linux/hrtimer.h**:

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

void hrtimer_init   (struct hrtimer *timer, clockid_t which_clock, enum hrtimer_mode mode);
int hrtimer_start   (struct hrtimer *timer, ktime_t time, enum hrtimer_mode mode);
unsigned long hrtimer_forward(struct hrtimer *timer, ktime_t now, ktime_t interval);

int hrtimer_cancel        (struct hrtimer *timer);
```



```
int hrtimer_try_to_cancel (struct hrtimer *timer);

int hrtimer_restart(struct hrtimer *timer);

ktime_t hrtimer_get_remaining (const struct hrtimer *timer);
int hrtimer_active (const struct hrtimer *timer);
int hrtimer_get_res (const clockid_t which_clock, struct timespec *tp);
```

The only field in the data structure:

```
struct hrtimer {
    struct rb_node          node;
    ktime_t                 expires;
    enum hrtimer_restart   (*function)(struct hrtimer *);
    struct hrtimer_base     *base;
};
```

that needs to be set by the user is function().

The earliest implementation of the API had a void argument to the function and also a data field in the structure. With the current API, one will probably want to embed the timer structure in a data structure that can be used to pass data into the function.

This can be done with the container_of() macro as such:

```
static struct my_data {
     struct hrtimer timer;
     other data ;
}
struct timer *my_timer;
struct my_data *dat = container_of (my_timer, struct my_data, timer);
```

where the first argument is a pointer to the timer structure, the second the type of structure it is contained in, and the third is the name of the timer structure in the data structure.

The return value of the function should be HRTIMER_NORESTART for a one-shot timer, and HRTIMER_RESTART for a recurring timer.

For the recurring case, the function hrtimer_forward() should be called to reset a new expiration time before the callback function returns. The now argument should be the current time. It can be obtained with:

```
struct hrtimer *timer;
....
ktime_t now = timer->base->get_time();
```

A hrtimer is initialized by hrtimer_init() and is bound to the type of clock specified by which_clock which can be CLOCK_MONOTONIC or CLOCK_REALTIME which matches current real-world time, and can differ if the system time is altered, such as by network time protocol daemons.

Once initialized the timer is launched with hrtimer_start(). If mode = HRTIMER_MODE_ABS the argument time is absolute; if mode = HRTIMER_MODE_REL it is relative.

The function hrtimer_cancel() will wait until the timer is no longer active, and its function is not running on any CPU, returning 0 if the timer has already expired and 1 if it was successfully canceled. The hrtimer_try_to_cancel() function differs but won't wait if the function is currently running, and will return -1 in that case. A canceled timer can be restarted by calling hrtimer_restart().

The remaining functions return the remaining time before expiration, whether the timer is currently on the queue, and ascertain the clock resolution in nanoseconds.

Here's an example of simple high resolution timer:

```
#include <linux/module.h>
#include <linux/timer.h>
#include <linux/init.h>
#include <linux/version.h>
#include <linux/ktime.h>
#include <linux/hrtimer.h>

static struct kt_data
{
    struct hrtimer timer;
    ktime_t period;
} *data;

static enum hrtimer_restart ktfun (struct hrtimer *var)
{
    ktime_t now = var->base->get_time ();
    printk (KERN_INFO "timer running at jiffies=%ld\n", jiffies);
    hrtimer_forward (var, now, data->period);
    return HRTIMER_NORESTART;
}
static int __init my_init (void)
{
    data = kmalloc (sizeof (*data), GFP_KERNEL);
    data->period = ktime_set (1, 0);    /* short period, 1 second  */
    hrtimer_init (&data->timer, CLOCK_REALTIME, HRTIMER_MODE_REL);
    data->timer.function = ktfun;
    hrtimer_start (&data->timer, data->period, HRTIMER_MODE_REL);

    return 0;
}
static void __exit my_exit (void)
{
    hrtimer_cancel (&data->timer);
    kfree (data);
}

module_init (my_init);
module_exit (my_exit);
MODULE_LICENSE ("GPL v2");
```


11.9 Labs

Lab 1: Kernel Timers from a Character Driver

Write a driver that puts launches a kernel timer whenever a write() to the device takes place.

Pass some data to the driver and have it print out.

Have it print out the current->pid field when the timer function is scheduled, and then again when the function is executed.

Lab 2: Multiple Kernel Timers

Make the period in the first lab long enough so you can issue multiple writes before the timer function run. (Hint: you may want to save your data before running this lab.)

How many times does the function get run?

Fix the solution so multiple timers work properly.

Lab 3: Periodic Kernel Timers

Write a module that launches a periodic kernel timer function; i.e., it should re-install itself.

Lab 4: Multiple Periodic Kernel Timers

Write a module that launches two periodic kernel timer functions; i.e., they should re-install themselves.

One periodic sequence should be for less than 256 ticks (so it falls in the tv1 vector), and the other should be for less than 16 K ticks (so it falls in the tv2 vector.)

Each time the timer functions execute, print out the total elapsed time since the module was loaded (in jiffies).

For one of the functions, also read the **TSC** and calibrate with the CPU frequency (as read from /proc/cpuinfo or the cpu_khz variable) to print out the elapsed time (hopefully) more accurately.

Lab 5: High Resolution Timers

Do the same things as in the previous exercise, setting up two periodic timers, but this time use the **hrtimer** interface.

Lab 6: Using kprobes to get statistics.

Using **kprobes**, find out how often kernel timers are deleted before they are run.

Examination of the kernel source discloses that the exported function `__mod_timer()` is called every time either `add_timer()` or `mod_timer()` is called.

You can see how often timers are deleted by monitoring `del_timer()` and `del_timer_sync()`; however, on single processor systems, `del_timer_sync()` is not defined.

Timers are frequent so you'll probably won't want to print out every time they are scheduled or deleted, but say every 100 times plus final statistics.

Is it possible that timer deletion can be more frequent than timer scheduling?

Lab 7: Mutex Locking from a Timer.

Write a simple module that loads a timer and takes out a mutex and then releases it when the timer runs.

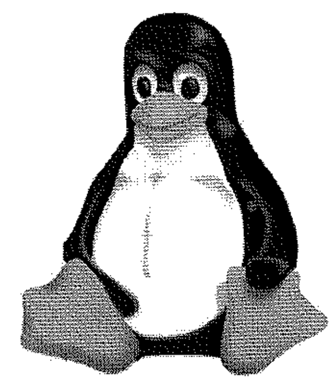
Doing this in an interrupt handler is supposed to be illegal. Here we have a softirq context; is that illegal too? Is this ignored, enforced, or warned against?

Lab 8: Executing a process from a timer.

Modify your first lab so the long period timer executes a user process, such as **wall**.

Chapter 12

Race Conditions and Synchronization Methods



We'll consider some of the methods the kernel uses to syncrhronize and avoid race conditions. We'll discuss atomic functions and bit operations, the use of **spinlocks**, **mutexes**, **semaphores**, and **completion functions**. Finally we'll see how the kernel maintains reference counts.

12.1 Concurrency and Synchronization Methods

Kernel execution is asynchronous and unpredictable; interrupts occur at any time, system calls can be entered from many different processes, and kernel threads of execution will also occupy the CPUs.

Many kernel resources can be modified in one place while being used in another. In some cases the code paths are distinct, while in others the same code is being executed more than once simultaneously. Either way data corruption is a danger as are potential race conditions including deadlock.

Such **concurrency** can be of two types:

- **True concurrency** occurs on **SMP** systems, when two threads of execution on different processors simultaneously access a resource.
- **Pseudo-concurrency** occurs even on single processor systems, when one thread is pre-empted or interrupted and another accesses an open resource.

A variety of mechanisms can be used to ensure integrity of shared resources; let's consider the various methods in order of **increasing** overhead.

The simplest method is the use of **atomic functions**, which work on specially typed variables which are essentially integers and include the use of **atomic bit operations**. Atomic functions:

- Execute in one single instruction; i.e., they can not be interrupted in mid-stream, and if two operations are requested simultaneously one must complete before the second can proceed.
- Can be used either in or out of process context.
- Can never go to sleep.
- Do not suspend interrupts while executing.

If more than one operation needs to be performed, one can use **spinlocks**; these get their name because if one attempts to take out a spinlock which is already held, the code will **spin**; i.e., do a busy wait, until the lock is available. The spinlock functions:

- Can be used either in or out of process context, but if used in interrupts, the forms which temporarily block interrupts should be used when the same spinlock is referenced in process context.
- Can block but do not go to sleep; i.e., another process can not be scheduled in.
- Can suspend interrupts while being used.
- Have supplemental **read** and **write** forms for the case in which one wants to permit simultaneous readers, and writes are relatively rare.

If one wants the ability to go to sleep if the resource is not available (to call the scheduler and have it yield the CPU to another process) one can use either **mutexes** (for which the basic operations are `mutex_lock()` and `mutex_unlock()`) or **semaphores**, (for which the basic operations are `up()` and `down()`.) Whenever possible mutexes should be used rather than semaphores in new code.

One should note:

- One can use `mutex_lock()` and `down()` only in process context; `mutex_unlock()` should not be used from an interrupt context, while the `up()` functions can be be used at any time.
- The `mutex_lock()` and `down()` functions can sleep. The sleep may or may not be interruptible by signals, depending on the form used.
- Interrupts are not suspended by these functions.
- Like spinlocks, the semaphore functions also have supplemental **read** and **write** forms for the case in which one wants to permit simultaneous readers when writing is relatively rare.

As an alternative, **completion** functions can be used in place of semaphores in the case where contention is expected to be high.

The kernel also employs the **seqlock()** mechanism, used when one often has to read a value which is rarely changed, and for which speed is essential.

Another method for similar situations is **RCU** (for Read-Copy-Update), which can lead to great performance boosts.

Exactly which mechanism should be used depends on:

- Whether contention is expected to be high or low.
- Whether one is in or out of process context.
- How many operations have to be performed while the lock is held.
- Whether sleeping is permissible.
- How often the lock needs to be taken.

For frequent operations, one would generally pick the one with the lowest overhead which fits the other requirements.

12.2 Atomic Operations

Atomic functions (many of which are macros) are completed as one single instruction and work on a variable of type `atomic_t`, which is a structure defined as

```
typedef struct {
    volatile int counter;
} atomic_t;
```

Using a structure helps prevent mixing up atomic variables with normal integers as you can't use atomic functions on integers and vice versa without explicit casting.

These in-line macros and functions are **SMP**-safe and depend on the architecture:

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

#define ATOMIC_INIT(i)    { (i) }
#define atomic_read(v)    ((v)->counter)
#define atomic_set(v,i)   (((v)->counter) = (i))

void atomic_add (int i, atomic_t *v);
void atomic_sub (int i, atomic_t *v);
void atomic_inc (atomic_t *v);
void atomic_dec (atomic_t *v);

int atomic_dec_and_test (atomic_t *v);
int atomic_inc_and_test_greater_zero (atomic_t *v);
int atomic_sub_and_test (int i, atomic_t *v);
int atomic_add_negative (int i, atomic_t *v);
int atomic_sub_return (int i, atomic_t *v);
int atomic_add_return (int i, atomic_t *v);
int atomic_inc_return (int i, atomic_t *v);
int atomic_dec_return (int i, atomic_t *v);
```

Note that the `ATOMIC_INIT()`, `atomic_read()`, and `atomic_set()` macros are automatically atomic since they just read a value.

On 64-bit platforms there are also a 64-bit atomic type and associated functions, such as `void atomic64_inc(atomic64_t *v)`. You can see the appropriate header file for details.

12.3 Bit Operations

In order to examine and modify individual bits in various flag and lock variables there are a number of atomic **bit** operation functions provided by the kernel.

These are accomplished through a single machine operation and thus are very fast; on most platforms this can be done without disabling interrupts.

The functions, not surprisingly, differ somewhat according to architecture. They are defined in **/usr/src/linux/arch/x86/include/asm/bitops.h**:

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

void set_bit            (int nr, volatile unsigned long *addr);
void clear_bit          (int nr, volatile unsigned long *addr);
void change_bit         (int nr, volatile unsigned long *addr);

int test_bit            (int nr, volatile unsigned long *addr);
int test_and_set_bit    (int nr, volatile unsigned long *addr);
int test_and_clear_bit  (int nr, volatile unsigned long *addr);
int test_and_change_bit (int nr, volatile unsigned long *addr);
```



```
long find_first_zero_bit (const unsigned long *addr, unsigned long size);
long find_next_zero_bit (const unsigned long *addr, long size, long offset);
long find_first_bit ( const unsigned long *addr, unsigned long size);

unsigned long ffz (unsigned long word);
unsigned long ffs (int x);
unsigned long fls (int x);
```

In these functions the type of **nr** depends on the architecture; for 32-bit **x86** it is just an integer. The second argument points to the variable in which bits are going to be examined or modified. It's type varies among architectures, usually being `volatile unsigned long *`.

The `test_` functions give the previous bit value as their return value.

There also exist a set of **non-atomic** bit functions, which differ from the above by being prefixed with `__`; e.g., `__set_bit()`. These can be used when locks are already taken out and integrity is assured, and are somewhat faster than the atomic versions.

12.4 Spinlocks

A **spinlock** is a mechanism for protecting critical sections of code. It will *spin* while waiting for a resource to be available, and not go to sleep.

One can protect the same code section from executing on more than one CPU, but more generally one protects simultaneous access to the same resource, which may be touched by differing code paths, which in addition, may be in or out of process context.

Spinlocks were important only on multi-processor systems before kernel preemption was included. This was because on **SMP** systems two CPUs can try to access a critical section of code simultaneously. Thus before kernel preemption was incorporated in the 2.6 kernel, on single processor systems, spinlocks were defined as no-ops. However, with a preemptible, hyper-threaded, or multi-core system, spinlocks are always operative.

The macros in **/usr/src/linux/arch/x86/include/asm/spinlock.h** (included from **/usr/src/linux/include/linux/spinlock.h**, when on an **SMP** system) contain the basic code for spinlocks. In the simplest invocation you have something like:

```
spinlock_t my_lock;
spin_lock_init (&my_lock);

spin_lock(&my_lock);
...... critical code ........
spin_unlock(&my_lock);
```

This guarantees the code touching the critical resource can't be run on more than one processor simultaneously, and does nothing on a single processor system with a non-preemptable kernel. However, the above functions should not be used out of process context (i.e., in interrupt handlers) as they may cause deadlocks in that case. (See **/usr/src/linux/Documentation/spinlocks.txt** for some further explanation.)

Often one wants to suspend, or disable, interrupt handling at the same time. In this case one does:

```
unsigned long flags;
spinlock_t my_lock;

spin_lock_init (&my_lock);

spin_lock_irqsave(&my_lock,flags);
...... critical code ........
spin_unlock_irqrestore(&my_lock,flags);
```

On single processor systems, this is not a no-op, as the interrupt disabling and restoring still goes on, and as mentioned these functions should be used when out of process-context. These functions take more time than the above "irq-less" versions.

Note that the disabling of interrupts occurs **only** on the current processor; other CPUs are free to handle interrupts while the lock is held.

There also exists the somewhat faster functions:

```
spin_lock_irq(&my_lock);
spin_unlock_irq(&my_lock);
```

the difference being that the original mask of enabled interrupts is not saved, and all interrupts are restored in the unlocking operation. This is dangerous (an interrupt may have been disabled) and generally these functions should not be used.

There are also reader and writer spinlock functions; with these there can be more than one reader in a critical region, but in order to make changes an exclusive write lock must be invoked. In other words, read lock blocks only a write lock, while a write lock blocks everyone. Examples would be:

```
unsigned long flags;
rwlock_t my_lock;
rw_lock_init (&my_lock);
....
read_lock_irqsave(&my_lock,flags);
...... critical code , reads only ..........
read_unlock_irqrestore(&my_lock,flags);

write_lock_irqsave(&my_lock,flags);
...... critical code , exclusive read and write access ....
write_unlock_irqrestore(&my_lock,flags);
```

There are also faster "irq-less" versions of these calls for non-interrupt contexts.

These locks favor readers over writers; i.e., if a writer is waiting for a lock and more readers come they will get first access. This can cause **writer starvation** and helped motivate the development and use of **seqlocks**.

There are a few other spinlock functions:

```
spin_unlock_wait(spinlock_t *lock);
int spin_is_locked(spinlock_t *lock);
int spin_trylock(spinlock_t *lock);
```


`spin_unlock_wait()` waits until the spinlock is free.

`spin_is_locked()` returns 1 if the spinlock is set.

`spin_trylock()` returns 1 if it got the lock; otherwise it returns with 0; i.e., it is a non-blocking call.

12.5 Big Kernel Lock

One locking mechanism used abundantly throughout the **Linux** kernel is the so-called **Big Kernel Lock**, or **BKL**. It is invoked simply with

```
lock_kernel();
....
critical code
....
unlock_kernel();
```

The **BKL** was originally a normal spinlock with widespread usage. As such, it is a relic of earlier times when locking was very coarse-grained.

However, in the 2.6.11 kernel the **BKL** was converted to a semaphore and really should be called the **Big Kernel Semaphore**. As a result it differs in two ways from other locking mechanisms:

- It can be applied recursively within a given thread.
- Sleeping is permitted while holding the **BKL**; it is released when sleep begins and grabbed again upon awakening.

Furthermore, with kernel preemption turned on it is possible to also turn on **BKL** preemption as a kernel configuration option through kernel version 2.6.24; after that it is automatically done.

Newcomers are sometimes confused by code like:

```
lock_kernel();
....
spin_lock(&my_lock);
....
spin_unlock(&my_lock);
....
unlock_kernel();
```

in which a finer-grained lock is nested within the **BKL**. The confusion arises because of not understanding that all spinlocks are **advisory**; i.e., they only are effective when code examines a lock status. They are not **mandatory** locks.

There is an ongoing effort to exterminate almost all instances of the **BKL**, as its promiscuous use leads to a lot of bottlenecks when code that in no way interacts with other code takes the **BKL** and suspends the other code.

Minimizing the use of the **BKL** is tedious. Each removal requires careful examination and testing against unanticipated side effects, but it is a necessary chore to accomplish on the way to fully fine-grained locking. Removal involves replacing it with appropriate and narrow locking mechanisms for each particular purpose.

Most likely the **BKL** will not disappear completely; rather it will be retained for some critical functions, especially for ones where time of execution is not critical, such as loading modules, system start up, etc.

12.6 Mutexes

A **mutex** (**mut**ual **ex**clusion object) is a basic kind of sleepable locking mechanism. While **spinlocks** are also a kind of mutex, they do not permit sleeping.

The elementary data structure is defined in **/usr/src/linux/include/linux/mutex.h**:

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

struct mutex {
   atomic_t          count;
   spinlock_t        wait_lock;
   struct list_head  wait_list;
};
```

and is meant to be used opaquely. If count is 1, the mutex is free, if it is 0 it is locked, and if it is negative, it is locked and processes are waiting.

Mutexes are initialized in an unlocked state with

```
DEFINE_MUTEX (name);
```

at compile time or

```
void mutex_init (struct mutex *lock);
```

at run time.

The locking primitives come in uninterruptible and interruptible forms but with only one unlocking function:

```
void mutex_lock (struct mutex *lock);
int mutex_lock_interruptible (struct mutex *lock);
int mutex_lock_killable (struct mutex *lock);

void mutex_unlock (struct mutex *lock);
```

Any signal will break a lock taken out with `mutex_lock_interruptible()` while only a fatal signal will break one taken out with `mutex_lock_killable()`. Locks taken out by `mutex_lock()` are not affected by signals.

There are some important restrictions on the use of mutexes:

- The mutex must be released by the original owner.
- The mutex can not be applied recursively.
- The mutex can not be locked or unlocked from interrupt context.

Note however, that violation of these restrictions will not be detected without mutex debugging configured into the kernel.

The **owner** of the mutex is the task in whose context the mutex is taken out. If that task is no longer active another task may release the mutex without triggering debugging warnings.

A non-blocking attempt to get the lock can be made with

```
int mutex_trylock (struct mutex *lock);
```

and

```
int mutex_is_locked (struct mutex *lock);
```

checks the state of the lock.

Note that `mutex_trylock()` returns 1 if it obtains the lock, akin to `spin_trylock()` and unlike `down_trylock()`.

There are no special reader/writer mutexes such as there are for other exclusion devices such as semaphores and spinlocks.

12.7 Semaphores

It is also possible to use counting **semaphores** to protect critical sections of code. These work on data structures of type `semaphore` and `rw_semaphore`.

The basic functions (defined as macros) can be found in **/usr/src/linux/include/linux/semaphore.h**, and **/usr/src/linux/include/linux/rwsem.h**:

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

void down (struct semaphore *sem);
int down_interruptible (struct semaphore *sem);
int down_trylock (struct semaphore *sem);
void up (struct semaphore *sem);

void down_read (struct rw_semaphore *sem);
void down_write (struct rw_semaphore *sem);
void up_read (struct rw_semaphore *sem);
void up_write (struct rw_semaphore *sem);

struct semaphore {
        atomic_t count;
        int sleepers;
```

```
        wait_queue_head_t wait;
};
struct rw_semaphore {
        signed long             count;
        spinlock_t              wait_lock;
        struct list_head        wait_list;
};
```

The `down()` function checks to see if someone else has already entered the critical code section; if the value of the semaphore is greater than zero, it decrements it and returns. If it is already zero, it will sleep and try again later.

The `down_interruptible()` function differs in that it can be interrupted by a signal; the other form blocks any signals to the process, and should be used only with great caution. However, you will now have to check to see if a signal arrived if you use this form, so you'll have code like:

```
if (down_interruptible(&sem)) return -ERESTARTSYS
```

which tells the system to either retry the system call or return `-EINTR` to the application; which it does depends on how the system is set up.

The `down_trylock()` form checks if the semaphore is available, and if not returns, and is thus a non-blocking `down` function (which is why it doesn't need an interruptible form.) It returns `0` if the lock is obtained. For instance, a typical read entry from a driver might contain:

```
....
if (file->f_flags & O_NONBLOCK) {
        if (down_trylock(&iosem))
            return -EAGAIN;
} else
        if (down_interruptible(&iosem))
            return -ERESTARTSYS;
}
```

The `up()` function increments the semaphore value, waking up any processes waiting on the semaphore. It doesn't require any `_interruptible` form.

The `_read, write` forms give finer control, permitting more than one reader to access the protected resource, but only one writer.

You have to be careful with semaphores; you can't lower them anywhere where sleeping would be very bad (such as in an interrupt routine) and you have to think things through carefully to avoid race conditions.

Semaphores may be declared and initialized with the following macros:

```
DECLARE_MUTEX(name);
DECLARE_RWSEM(name);
```

where `name` is an object of type `structsemaphore`.

The value of the semaphore can be directly manipulated with the inline function:


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

and is often initialized with the following inline convenience functions:

```
static inline void init_MUTEX (struct semaphore *sem){
        sema_init(sem, 1);
}
static inline void init_MUTEX_LOCKED (struct semaphore *sem){
        sema_init(sem, 0);
}
```

- Historically, semaphores have more often been used as binary **mutexes** rather than as counters.
- A semaphore may be more difficult to debug than a mutex in that it can have more than one owner at a time.
- Any new code should use mutexes unless the counting capability of semaphores is really needed. The migration of most existing semaphores to mutexes is gradually and carefully being done.

An example, culled from **/usr/src/linux/kernel/sys.c**:

```
....
2.6.31:1116 DECLARE_RWSEM(uts_sem);
....
2.6.31:1129 SYSCALL_DEFINE2(sethostname, char __user *, name, int, len)
2.6.31:1130 {
2.6.31:1131     int errno;
2.6.31:1132     char tmp[__NEW_UTS_LEN];
2.6.31:1133
2.6.31:1134     if (!capable(CAP_SYS_ADMIN))
2.6.31:1135             return -EPERM;
2.6.31:1136     if (len < 0 || len > __NEW_UTS_LEN)
2.6.31:1137             return -EINVAL;
2.6.31:1138     down_write(&uts_sem);
2.6.31:1139     errno = -EFAULT;
2.6.31:1140     if (!copy_from_user(tmp, name, len)) {
2.6.31:1141             struct new_utsname *u = utsname();
2.6.31:1142
2.6.31:1143             memcpy(u->nodename, tmp, len);
2.6.31:1144             memset(u->nodename + len, 0, sizeof(u->nodename) - len);
2.6.31:1145             errno = 0;
2.6.31:1146     }
2.6.31:1147     up_write(&uts_sem);
2.6.31:1148     return errno;
2.6.31:1149 }
....
```

```
2.6.31:1153 SYSCALL_DEFINE2(gethostname, char __user *, name, int, len)
2.6.31:1154 {
2.6.31:1155     int i, errno;
2.6.31:1156     struct new_utsname *u;
2.6.31:1157
2.6.31:1158     if (len < 0)
2.6.31:1159             return -EINVAL;
2.6.31:1160     down_read(&uts_sem);
2.6.31:1161     u = utsname();
2.6.31:1162     i = 1 + strlen(u->nodename);
2.6.31:1163     if (i > len)
2.6.31:1164             i = len;
2.6.31:1165     errno = 0;
2.6.31:1166     if (copy_to_user(name, u->nodename, i))
2.6.31:1167             errno = -EFAULT;
2.6.31:1168     up_read(&uts_sem);
2.6.31:1169     return errno;
2.6.31:1170 }
```

Kernel Version Note **Kernel Version Note**

- As of kernel 2.6.26 the effort to remove as many semaphores as possible has accelerated. It is possible they will be removed altogether at some point, with the last remaining cases converted to the **completion** API.
- As part of this effort, the architecture-dependent code is being replaced with generic C code, which reduces the code base.
- In kernel 2.6.26 the type of the **count** field in the **semaphore** structure was changed to `unsigned int`.

12.8 Completion Functions

The completion functions give an alternative method of waiting for events to take place, and are meant to be used in place of semaphores in some places.

This API is optimized to work in the case of contention, while semaphores are optimized to work in the case of non-contention, and thus is somewhat more efficient.

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

struct completion {
```



```
    unsigned int done;
    wait_queue_head_t wait;
};

void init_completion (struct completion *c);
void wait_for_completion (struct completion *c);
int  wait_for_completion_interruptible (struct completion *c);
void complete (struct completion *c);
void complete_and_exit (struct completion *c, long code);

unsigned long wait_for_completion_timeout (struct completion *c, unsigned long timeout);
unsigned long wait_for_completion_interruptible_timeout (struct completion *c,
                                                         unsigned long timeout);
```

Note the versions with **timeout** in their name will return without a wake up call if the expiration period is reached without one. Both these and the non-interruptible forms have non-void return values and need to be checked.

The completion structure can be declared and initialized in either of two ways:

```
DECLARE_COMPLETION(x);

struct completion x;
init_completion(&x);
```

The `complete_and_exit()` function doesn't return if successful, takes an additional argument, `code`, which is the exit code for the kernel thread which is terminating. Obviously, this function should only be used in cases such as terminating a background thread, as it will kill whatever it is doing.

The correspondence with semaphores is very simple; there is a one to one mapping between the methods:

```
struct semaphore  <--->  struct completion
sema_init()       <--->  init_completion
down()            <--->  wait_for_completion()
up()              <--->  complete()
```

12.9 Reference Counts

One often needs to maintain a reference counter for an object, such as a data structure. If a resource is used in more than one place and passed to various subsystems, such a reference count is probably required.

The **kref API** should be used for maintaining such reference counts, rather than having them constructed by hand. The relevant functions are defined in **/usr/src/linux/include/linux/kref.h**:

```
struct kref {
        atomic_t refcount;
};
```

```
void kref_init (struct kref *kref);
void kref_set  (struct kref *kref, int num);
void kref_get  (struct kref *kref);
int kref_put (struct kref *kref, void (*release) (struct kref *kref));
```

It is presumed that the kref structure is embedded in a data structure you are using, which can be the object you are reference counting, as in:

```
struct my_dev_data {
   ...
   struct kref my_refcount;
   ...
};
```

One must initialize with `kref_init()`, and can increment the reference count with `kref_get()`.

Decrementing the reference count is done with `kref_put()` and if the reference count goes to zero the function referred to in the second argument to `kref_put()` is called. This may not be `NULL` or just `kfree()` (which is explicitly checked for.) This is likely to be something like this:

```
struct my_dev_data *md;

kref_put(md->my_refcount, my_release);

static void my_release (struct kref *kr){
   struct my_dev_data *md = container_of (kr, struct my_dev_data, my_refcount);
   my_dev_free(md);
}
```

where one does whatever is necessary in `my_dev_free` to free up any memory and other resources.

Note the use of the `container_of()` macro, which does pointer arithmetic; its first argument is the pointer we are given, the second the type of structure it is embedded in, and the third is its name in that structure. The result returned is a pointer to that structure.

A good guide to using kernel reference counts can be found in **/usr/src/linux/Documentation /kref.txt**.

12.10 Labs

Lab 1: Semaphore Contention

Write three simple modules where the second and third one use a variable exported from the first one. The second and third one can be identical; just give them different names.

Hint: You can use the macro `__stringify(KBUILD_MODNAME)` to print out the module name.

You can implement this by making small modifications to your results from the modules exercise.

The exported variable should be a semaphore. Have the first module initialize it in the unlocked state.

The second (third) module should attempt to lock the semaphore, and if it is locked, fail to load; make sure you return the appropriate value from your initialization function.

Make sure you release the semaphore in your cleanup function.

Test by trying to load both modules simultaneously, and see if it is possible. Make sure you can load one of the modules after the other has been unloaded, to make sure you released the semaphore properly.

Lab 2: Mutex Contention

Now do the same thing using **mutexes** instead of semaphores.

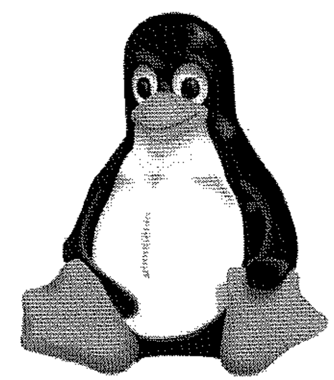
Lab 3: Mutex Unlocking from an Interrupt.

Modify the simple interrupt sharing lab to have a mutex taken out and then released in the interrupt handler.

This is supposed to be illegal. Is this ignored, enforced, or warned against? Why?

Chapter 13

ioctls



We'll consider the **ioctl** method (I/O Control) which is a grab bag which can be used in many different ways for applications to communicate with device drivers. We discuss what **ioctl**'s are, how they are called, and how to write driver entry points for them.

13.1 What are ioctls?

ioctl's (input output control) are special functions which are unique to a device or class of device. `ioctl()` is both a call from user-space, as well as a driver entry point (i.e., like `write()`, `read()`, etc.)

Various **commands** can be implemented which either send to or receive information from a device. One can control device driver behaviour; i.e, shutdown, reset and modify. One can send out-of-band messages even while reads and writes are pending.

Excessive use of **ioctl**'s is not favored by **Linux** kernel developers, as by their very nature they can be used to do almost anything, including adding what are essentially new system calls.

To use **ioctl**'s, one has to first open a device using the `open()` system call, and then send the appropriate `ioctl()` command and any necessary arguments.

```
#include <sys/ioctl.h>

int ioctl(int fd, int command, ...);
```

The third argument is usually written as `char *argp`. The use of the dots usually means a variable number of arguments. Here it indicates that type checking should not be done on the argument, so we are utilizing a trick. You shouldn't pass more than three arguments to the `ioctl()` call.

On success 0 is returned, and on error -1 is returned with `errno` set. The possible error returns are:

Table 13.1: **ioctl() return values**

Value	Meaning
EBADF	Bad file descriptor.
ENOTTY	File descriptor not associated with a character special device, or the request does not apply to the kind of object the file descriptor references.
EINVAL	Invalid command or argp.

Example:

```
int fd = open ("/dev/mydrvr", O_RDWR);

if ( ioctl( fd, MYDRVR_SET, buf) < 0)
       perror( "MYDRVR_SET ioctl failed" );
```

13.2 Driver Entry point for ioctls

The entry point for `ioctl()` looks like:

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

static int mydrvr_ioctl (struct inode *inode, struct file *file, unsigned int cmd,
                         unsigned long arg);
```


where `arg` can be used directly either as a `long` or a pointer in user-space. In the latter case, the proper way to is though the `put_user()`, `get_user()`, `copy_to_user()`, `copy_from_user()` functions.

Example:

```
static int mydrvr_ioctl (struct inode  *inode, struct file   *file, unsigned int  cmd,
                          unsigned long arg)
{
   if (_IOC_TYPE(cmd) != MYDRBASE)
     return (-EINVAL);

   switch (cmd) {

   case MYDRVR_RESET :
   /* do something */
      return 0;

   case MYDRVR_OFFLINE :
   /* do something */
      return 0;

   case MYDRVR_GETSTATE :
      if (copy_to_user ((void *)arg, &mydrvr_state_struct, sizeof(mydrvr_state_struct)))
           return -EFAULT;
      return 0;

   default:
        return -EINVAL;
   }
}
```

13.3 Lockless ioctls

Normal `ioctl()` calls operate under the **BKL** (Big Kernel Lock), and if they take a long time to run can cause large latencies in potentially unrelated areas.

The `ioctl()` system call passes through `sys_ioctl()`, which calls `vfs_ioctl()` in **/usr/src/linux /fs/ioctl.c**:

```
static long vfs_ioctl(struct file *filp, unsigned int cmd,
                      unsigned long arg)
{
    int error = -ENOTTY;

    if (!filp->f_op)
            goto out;

    if (filp->f_op->unlocked_ioctl) {
            error = filp->f_op->unlocked_ioctl(filp, cmd, arg);
            if (error == -ENOIOCTLCMD)
```

```
                error = -EINVAL;
            goto out;
    } else if (filp->f_op->ioctl) {
            lock_kernel();
            error = filp->f_op->ioctl(filp->f_path.dentry->d_inode,
                                      filp, cmd, arg);
            unlock_kernel();
    }

out:
    return error;
}
```

which shows that if the method

```
long unlocked_ioctl (struct file *filp, unsigned int cmd, unsigned long arg);
```

is defined, it will supersede the unlocked variety.

Note, however, the BKL is allowed to sleep unlike other locks and thus joining a wait queue is not enough to bottle up the system; you actually have to do some work which takes some CPU time to do so.

All new kernel code should use the lockless call, introduced in kernel 2.6.11, and old code should gradually be converted unless there is a known and good reason to take out the BKL.

- An active project is going underway to move the BKL out of the innards of the system call and into the particular drivers with the goal of one-by-one elimination.
- It also should be noted that `ioctl()` is not the only entry point that takes out the BKL in character drivers; in particular so do `open()` and `fsync()`. Another project is ongoing to eliminate the BKL from all of these entry points unless there is a true need for them.

13.4 Defining ioctls

Before using `ioctl()` one must choose the numbers corresponding to the integer `command` argument. Just picking arbitrary numbers is a bad idea; they should be unique across the system.

There are at least two ways errors could arise:

- Two device nodes may have the same major number.
- An application could make a mistake, opening more than one device and mixing up the file descriptors, thereby sending the right command to the wrong device.

Results might be catastrophic and even damage hardware.

Two important files are **/usr/src/linux/include/asm-generic/ioctl.h** and **/usr/src/linux/Documentation/ioctl-number.txt**.

In the present implementation `command` is 32 bits long; the command is in the lower 16 bits (which was the old size.) There are four bit-fields:

Table 13.3: **ioctl() command bit fields**

Bits	Name	Meaning	Description
8	_IOC_TYPEBITS	type	magic number to be used throughout the driver
8	_IOC_NRBITS	number	the sequential number
14	_IOC_SIZEBITS	size	of the data transfer
2	_IOC_DIRBITS	direction	of the data transfer

The direction can be one of the following:

```
_IOC_NONE
_IOC_READ
_IOC_WRITE
_IOC_READ | _IOC_WRITE
```

and is seen from the point of view of the *application*.

The size and direction information can be used to simplify sending data back and forth between user-space and kernel-space, using `arg` as a pointer. Note there is no enforcement here; you can send information either way no matter what direction is used to define the command. The largest transfer you can set up this way is 16 KB, since you have only 14 bits available to encode the size.

You are not required to pay attention to the split up of the bits in the command, but it is a good idea to do so.

Useful macros:

Encode the `ioctl` number:

```
_IO(  type, number)
_IOR( type, number, size)
_IOW( type, number, size)
_IOWR(type, number, size)
```

One has to be careful about how the parameter `size` is used. Rather than passing an integer, one passes the actual data structure, which then gets a `sizeof()` primitive applied to it; e.g.,

```
MY_IOCTL = _IOWR('k', 1, struct my_data_structure);
```

This won't work if `my_data_structure` has been allocated dynamically, as in that case `sizeof()` will return the size of the pointer.

Decode the `ioctl` number:

```
_IOC_DIR( cmd)
_IOC_TYPE(cmd)
_IOC_NR(  cmd)
_IOC_SIZE(cmd)
```

Example:

```
#define MYDRBASE 'k'
#define MYDR_RESET _IO(  MYDRBASE, 1)
#define MYDR_STOP  _IO(  MYDRBASE, 2)
#define MYDR_READ  _IOR( MYDRBASE, 2, my_data_buffer)
```

13.5 Labs

Lab 1: Using ioctl's to pass data

Write a simple module that uses the **ioctl** directional information to pass a data buffer of fixed size back and forth between the driver and the user-space program.

The size and direction(s) of the data transfer should be encoded in the command number.

You'll need to write a user-space application to test this.

Lab 2: Using ioctl's to pass data of variable length.

Extend the previous exercise to send a buffer whose length is determined at run time. You will probably need to use the `_IOC` macro directly in the user-space program. (See `linux/ioctl.h`.)

Lab 3: Using ioctl's to send signals.

It is sometimes desirable to send a signal to an application from within the kernel. The function for doing this is:

```
int send_sig (int signal, struct task_struct *tsk, int priv);
```

where `signal` is the signal to send, `tsk` points to the task structure corresponding to the process to which the signal should be sent, and `priv` indicates the privilege level (0 for user applications, 1 for the kernel.)

Write a character driver that has three `ioctl` commands:

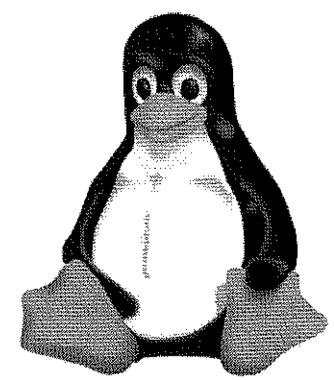
- Set the process ID to which signals should be sent.
- Set the signal which should be sent.
- Send the signal.

You'll also have to develop the sending program.

- If given no arguments it should send `SIGKILL` to the current process.
- If given one argument it should set the process ID to send signals to.
- If given two arguments it should also set the signal.

Chapter 14

The proc Filesystem



We'll discuss the **proc** pseudo-filesystem. We'll show how to create, destroy, read and write to entries in this filesystem. Finally we'll also discuss the **seq_file** interface and how it can be used to make **proc** filesystem entries.

14.1 What is the proc Filesystem?

The **proc** filesystem is a pseudo-filesystem; it exists only in memory and is mounted on an empty /proc directory. Information in the **proc** filesystem are generated only when it is accessed; it is not continually updated.

Entries in **proc** can be used to obtain information about the system (and device drivers) when read. Writing to entries can set system parameters and modify device functionality.

Two advantages of using **proc** (as opposed to using `ioctl()` calls or regular files) are that it is always available and is accessible to all users (providing the permissions are appropriate.)

Some features of **proc** have migrated over to the **sysfs** facility (mounted under `/sys`.) Since use of **proc** has always suffered from a lack of standards and conventions, one should always evaluate whether or not **proc** is the best choice.

14.2 Creating Entries

Creating, managing, and removing entries in the `proc` filesystem is done with:

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

struct proc_dir_entry *create_proc_entry (const char *name, mode_t mode,
                                          struct proc_dir_entry *parent);
void remove_proc_entry (const char *name, struct proc_dir_entry *parent);
struct proc_dir_entry *proc_symlink (const char *name, struct proc_dir_entry *parent,
                                     const char *dest);
struct proc_dir_entry *proc_mkdir (const char *name, struct proc_dir_entry *parent);
```

The `name` argument gives the name of the directory entry, which will be created with the permissions contained in the `mode` argument. If the `parent` argument is `NULL`, the entry will go in the `/proc` main directory.

The function `proc_symlink()` creates a symbolic link; it is equivalent to doing: `ln -s dest name`.

The function `proc_mkdir()` creates directory `name` under `parent`.

The parent directory can be something you created with `proc_mkdir()`, or if you want to put it in an already created subdirectory of `/proc`. Convenient ones include `/proc/driver`, `/proc/fs`, `/proc/net`, `/proc/net/stat`, `/proc/bus`, and `/proc/driver`. For instance one can do:

```
my_proc = create_proc_entry ("driver/my_proc", NULL, NULL);
```

The basic data structure here is `proc_dir_entry` which is given by:

```
struct proc_dir_entry {
        unsigned short low_ino;
        unsigned short namelen;
        const char *name;
        mode_t mode;
        nlink_t nlink;
        uid_t uid;
        gid_t gid;
        unsigned long size;
        struct inode_operations * proc_iops;
        struct file_operations * proc_fops;
        get_info_t *get_info;
        struct proc_dir_entry *next, *parent, *subdir;
        void *data;
        read_proc_t *read_proc;
```



```
        write_proc_t *write_proc;
        atomic_t count;         /* use count */
        int deleted;            /* delete flag */
        kdev_t  rdev;
};
```

The important fields we will discuss in detail are the read and write callback functions, `read_proc()`, `write_proc()`, which we'll discuss next.

The `data` pointer can be used like a `private_data` pointer. In particular it is useful for using the same read and write callback functions for multiple entries.

Note you can change the ownership and permission fields of the above structure and it will be reflected in the directory entry.

14.3 Reading Entries

When a process tries to read an entry in the `proc` filesystem, it causes invocation of the read callback function associated with the directory entry; i.e., you would have something like:

```
static struct proc_dir_entry *my_proc_entry;
...
   my_proc_entry = create_proc_entry ("my_proc", 0, NULL);
   my_proc_entry->read_proc = my_proc_read;
```

perhaps in `init_module()`, where the read callback function, `my_proc_read()` has been previously defined. This has an integer return type and its prototype definition is given by:

```
typedef int (read_proc_t)(char *page, char **start, off_t off, int count, int *eof,
                          void *data);
```

When someone tries to read the entry, the information will be written into the `page` argument at an offset of `off`, writing at most `count` bytes. For reading just a few bytes, the callback function usually ignores these arguments.

The `eof` argument is only used when `off` and `count` are used; it should signal the end of the file with a 1. The `start` argument is a left-over legacy from earlier implementations and isn't used. The `data` argument can be used to create a single callback function for multiple `proc` entries, or for other purposes.

When successful, your read function should return the number of bytes written into the buffer pointed to by `page`. Here's a simple example of a module using a `proc` read callback:

```
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/init.h>
#include <linux/version.h>
#include <linux/jiffies.h>

static int x_delay = 1;         /* the default delay */
```

```
static int
x_read_busy (char *buf, char **start, off_t offset, int len,
             int *eof, void *unused)
{
    unsigned long j = jiffies + x_delay * HZ;

    while (time_before (jiffies, j))
        /* nothing */ ;
    *eof = 1;
    return sprintf (buf, "jiffies = %d\n", (int)jiffies);
}

static struct proc_dir_entry *x_proc_busy;

static int __init my_init (void)
{
    x_proc_busy = create_proc_entry ("x_busy", 0, NULL);
    x_proc_busy->read_proc = x_read_busy;
    return 0;
}

static void __exit my_exit (void)
{
    if (x_proc_busy)
        remove_proc_entry ("x_busy", NULL);
}

module_init (my_init);
module_exit (my_exit);

MODULE_LICENSE ("GPL v2");
```

14.4 Writing Entries

When a process tries to write data to an entry in the proc filesystem, it causes invocation of the write callback function associated with the directory entry; i.e., you would have something like:

```
static struct proc_dir_entry *my_proc_entry;
...
   my_proc_entry = create_proc_entry ("my_proc", 0, NULL);
   my_proc_entry->write_proc = my_proc_write;
```

where the read callback function, `my_proc_write()` has been previously defined. This has an integer return type and its prototype definition is given by:

```
typedef int (write_proc_t)(struct file *file, const char __user *buffer, unsigned long count,
                           void *data);
```

This function will read **count** bytes (at most) from the location pointed to by **buffer**.

The **file** location is generally unused, and once again the location pointed to by the **data** argument can be used when a single callback function is used for multiple file entries or for other purposes.

It is important to note that **buffer** is a user space pointer; thus you must use a function like `copy_from_user()` to obtain its contents. Once you have the contents you can put them to use in your kernel functions as needed.

Note that usually `/proc` entries are **text**, not **binary**. This means to convert user-space input into usable form you may require the services of functions like `atoi()`. However, these are not defined in the kernel. Instead you need to use the following functions, defined in **/usr/src/linux/lib/vsprintf.c**:

```
         long simple_strtol (const char *cp,char **endp, unsigned int base);
unsigned long simple_strtoul (....);
unsigned long long simple_strtoull (....);
    long long simple_strtoll (....);
```

all of which have the same arguments. The first argument is a pointer to the string to convert, the second is a pointer to the end of the parsed string, and the third is the number base to use; giving `0` is the same as giving `10`. The following statements are equivalent:

```
long j = simple_strtol ("-1000", NULL, 10);
long j = simple_strtol ("-1000", 0, 0 );
```

You can also do the format conversion using the kernel implementation of `sscanf()`.

14.5 The seq_file Interface

The `seq_file` interface is often used for **read-only** entries in the **/proc** filesystem. It addresses the often encountered situation where data needs to be stored and/or printed as a series of sequential records. Thus **seq_file** is short hand for a pseudo **sequential file.**

Use of the relevant functions is particularly useful in maintaining (read-only) entries in the **proc** filesystem.

All the relevant structures and functions are contained in **/usr/src/linux/include/linux/seq_file.h** and **/usr/src/linux/fs/seq_file.c**. The first major data structure is the `seq_file` itself:

```
struct seq_file {
    char *buf;
    size_t size;
    size_t from;
    size_t count;
    loff_t index;
    struct semaphore sem;
    struct seq_operations *op;
};
```

Normally one need not work directly with the elements of this structure except to set the pointer to the jump table of operations for this structure;

```
struct seq_operations {
    void * (*start) (struct seq_file *m, loff_t *pos);
    void (*stop) (struct seq_file *m, void *v);
    void * (*next) (struct seq_file *m, void *v, loff_t *pos);
    int (*show) (struct seq_file *m, void *v);
};
```

The purpose of the `start()` method is to return a pointer to the member of the sequential series of items indicated by the value of the `*pos` argument; i.e., where the reading of the items should begin. This function may also need to establish some kind of lock to prevent corruption of the `seq_file` structure while it is being traversed.

The purpose of the `next()` method is to return a pointer to the **next** item; it should also increment `*pos`.

The purpose of the `stop()` method is to indicate the end of peeling off of the items. Usually it need do nothing more than release whatever lock has been acquired, if any.

The purpose of the `show()` method is to do the actual work of placing the data record in the `seq_file` data structure. It does this using the functions

```
static inline int seq_putc (struct seq_file *m, char c);
static inline int seq_puts (struct seq_file *m, const char *s);
int seq_printf (struct seq_file *, const char *, ...);
```

the use of which should be pretty clear from their close resemblance to the standard I/O functions, `putc()`, `puts()`, `printf()` and their variants. The output is delivered to the pseudo output stream represented by the `seq_file` structure pointer.

The reading of a `seq_file` data structure is usually done with the following standard methods, or file operations:

```
int seq_open (struct file *, struct seq_operations *);
ssize_t seq_read (struct file *, char *, size_t, loff_t *);
loff_t seq_lseek (struct file *, loff_t, int);
int seq_release (struct inode *, struct file *);
int seq_escape (struct seq_file *, const char *, const char *);
```

Of these, the one that usually requires substitution is the `open()` method. Typically one might have something like this:

```
static struct seq_operations my_seq_ops = {
    .start=   my_seq_start,
    .next=    my_seq_next,
    .stop=    my_seq_stop,
    .show=    my_seq_show,
}
static int my_seq_file_open (struct inode *inode, struct file *file){
    return ( seq_open( file, &my_seq_ops) );
}
static struct file_operations my_seq_file_ops = {
    .open=    my_seq_file_open,
```



```
    .read=    seq_read,
    .llseek=  seq_lseek,
    .release= seq_release,
}
```

You may be wondering how the file operations have access to the `seq_file` data structure; examination of the source code indicates the `private_data` field of the `file` structure is used for this purpose.

The `seq_file` interface is particularly useful for implementing **read-only** entries in the **proc** pseudo-filesystem. A good example can be found in **/usr/src/linux/drivers/pci/proc.c**, which maintains a number of **proc** entries.

A code fragment showing how to set this up for the `/proc/bus/pci` entry looks like:

```
....
static struct seq_operations proc_bus_pci_devices_op = {
    .start=    pci_seq_start,
    .next=     pci_seq_next,
    .stop=     pci_seq_stop,
    .show=     show_device
};
....
static int proc_bus_pci_dev_open(struct inode *inode,
                                 struct file *file)
{
    return seq_open(file, &proc_bus_pci_devices_op);
}
static struct file_operations proc_bus_pci_dev_operations = {
    .open=      proc_bus_pci_dev_open,
    .read=      seq_read,
    .llseek=    seq_lseek,
    .release=   seq_release,
};

  struct proc_dir_entry *entry;
  ....
  entry = create_proc_entry ("devices", 0, proc_bus_pci_dir);
  if (entry)
      entry->proc_fops = &proc_bus_pci_dev_operations;
  ....
```

You can look at the full source to get some examples of how the `start()`, `next()`, `stop()`, and `show()` methods are implemented.

14.6 Labs

Lab 1: /proc/kcore

Try to remove `/proc/kcore`. Is there is a permissions problem? If so try to reset them with `chmod 666 /proc/kcore` and try again.

If it doesn't work, explain what this file is, and why it is difficult (if not impossible) to remove.

> If you use **cat** to test your read entries in the following labs, you may find the unexpected behaviour that the entry point is always called twice, even when you signal end of file. (You can use **strace** to verify this happens with **all** proc entries and is not an error in your module.) This is due to the way **cat** is written and is nothing to worry about.

Lab 2: Using the /proc filesystem.

Write a module that creates a **/proc** filesystem entry and can read and write to it.

When you read from the entry, you should obtain the value of some parameter set in your module.

When you write to the entry, you should modify that value, which should then be reflected in a subsequent read.

Make sure you remove the entry when you unload your module. What happens if you don't and you try to access the entry after the module has been removed?

The solution shows how to create the entry in the `/proc` directory and also in the `/proc/driver` directory.

Lab 3: Making your own subdirectory in /proc.

Write a module that creates your own **proc** filesystem subdirectory and creates at least two entries under it.

As in the first exercise, reading an entry should obtain a parameter value, and writing it should reset it.

You may use the `data` element in the `proc_dir_entry` structure to use the same callback functions for multiple entries.

Lab 4: Using /proc to send signals.

It is sometimes desirable to send a signal to an application from within the kernel. The function for doing this is:

```
int send_sig (int signal, struct task_struct *tsk, int priv);
```

where `signal` is the signal to send, `tsk` points to the task structure corresponding to the process to which the signal should be sent, and `priv` indicates the privilege level (`0` for user applications, `1` for the kernel.)

Write a module that opens up two entries in the `proc` file system.

- When the first entry is written to, it sets the process ID of the process which is registered to receive signals via this mechanism.

- When the second entry is written to, it gets the signal to be delivered and then sends it.
- Reading either entry simply shows the current values of these parameters.

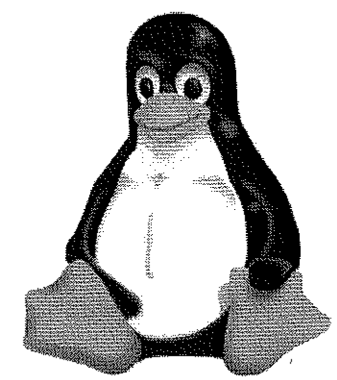
Lab 5: Using seq_file for the /proc filesystem.

Take the simple **x_busy proc** entry discussed earlier, and re-implement it using the `seq_file` interface.

As a parameter, input the number of lines to print out.

Chapter 15

Unified Device Model and sysfs



We'll consider the **unified device model**, it's main data structures and how they apply to real devices and examine the **sysfs** pseudo-filesystem.

15.1 Unified Device Model

A **unified device model** (or **integrated device model**) was introduced in the 2.6 kernel series. Under this scheme all devices are handled in one framework, with similar data structures and functional methods. Additionally, this framework is represented as a device tree rooted on the system buses.

For the most part, device drivers need not interact directly with this underlying model, but register as devices under the type of bus they are connected to, such as **pci**. Information about the devices is exposed in the **sysfs** filesystem, to which drivers can optionally export data for viewing, as a more modern alternative to the use of the **/proc** filesystem and **ioctl()** commands.

At the root of the driver model are **kobjects**, which contain simple representations of data related to any object in a system, such as a name, type, parent, reference count, lock, etc. A set of kobjects identical in type is contained in a **kset**.

The data structures incorporated in the new driver model contain information for each device such as what driver is used for them, what bus they are on, what power state they are in and how they suspend and resume. They also map out the structure of the system buses, how they are connected to each other and what devices can be attached and are attached.

15.2 Basic Structures

For every device there is a generic structure defined in **/usr/src/linux/include/linux/device.h**

```
struct device {
     struct device            *parent;

     struct device_private    *p;

     struct kobject kobj;
     const char               *init_name; /* initial name of the device */
     struct device_type       *type;

     struct semaphore         sem;   /* semaphore to synchronize calls to
                                      * its driver.
                                      */

     struct bus_type *bus;           /* type of bus device is on */
     struct device_driver *driver;   /* which driver has allocated this
                                        device */
     void            *driver_data;   /* data private to the driver */
     void            *platform_data; /* Platform specific data, device
                                        core doesn't touch it */
     struct dev_pm_info       power;

#ifdef CONFIG_NUMA
     int             numa_node;      /* NUMA node this device is close to */
#endif
     u64             *dma_mask;      /* dma mask (if dma'able device) */
     u64             coherent_dma_mask;/* Like dma_mask, but for
                                          alloc_coherent mappings as
                                          not all hardware supports
                                          64 bit addresses for consistent
                                          allocations such descriptors. */

     struct device_dma_parameters *dma_parms;

     struct list_head         dma_pools;     /* dma pools (if dma'ble) */

     struct dma_coherent_mem *dma_mem; /* internal for coherent mem
                                          override */

     /* arch specific additions */
     struct dev_archdata      archdata;

     dev_t                    devt;   /* dev_t, creates the sysfs "dev" */

     spinlock_t               devres_lock;
     struct list_head         devres_head;

     struct klist_node        knode_class;
     struct class             *class;
     struct attribute_group   **groups;        /* optional groups */

     void    (*release)(struct device *dev);
};
```


After important fields are initialized, the device is registered with and unregistered from the system core with:

```
int  device_register   (struct device *dev);
void device_unregister (struct device *dev);
```

Reference counts for the device are atomically incremented and decremented with:

```
struct device *get_device (struct device *dev);
void put_device (struct device *dev);
```

The pointer to a structure of type `device_driver` describes the **driver** for the **device**:

```
struct device_driver {
     const char               *name;
     struct bus_type          *bus;

     struct module            *owner;
     const char               *mod_name;     /* used for built-in modules */

     int (*probe) (struct device *dev);
     int (*remove) (struct device *dev);
     void (*shutdown) (struct device *dev);
     int (*suspend) (struct device *dev, pm_message_t state);
     int (*resume) (struct device *dev);
     struct attribute_group **groups;

     struct dev_pm_ops *pm;

     struct driver_private *p;
};
```

Drivers are registered/unregistered with the appropriate bus with:

```
int  driver_register   (struct device_driver *drv);
void driver_unregister (struct device_driver *drv);
```

and reference counts are incremented/decremented with:

```
struct device_driver *get_driver (struct device_driver *drv);
void                  put_driver (struct device_driver *drv);
```

Next we consider how this generic infrastructure connects to real devices.

15.3 Real Devices

Actual device drivers rarely work directly with the structures we have so far described; rather they are used by the internal code used for each specific type of bus and/or device.

For example, **PCI** devices have two important structures:

```
struct pci_dev {
.....
   struct pci_driver *driver;
....
   struct device dev;
.....
}

struct pci_driver {
.....
   struct device_driver driver;
....
}
```

and **drivers** are registered with the system with

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

int  pci_register_driver   (struct pci_driver *);
void pci_unregister_driver (struct pci_driver *);
```

Devices, on the other hand, are registered, or discovered, directly by the probe callback function or by `pci_find_device()`.

How is this connected with the generic infrastructure? Because the generic `device` structure is embedded in the `pci_dev` structure, and the generic `device_driver` structure is embedded in the `pci_driver` structure, one must do pointer arithmetic.

This is done through use of the macro `to_pci_dev()` as in:

```
struct device *dev;
.....
struct pci_dev *pdev = to_pci_dev (dev);
```

which is implemented in terms of the `container_of()` macro:


```
#define to_pci_dev(n) container_of(n, struct pci_dev, dev)
```

where the first argument is a pointer to the `device` structure, the second the type of structure it is contained in, and the third is the name of the `device` structure in the data structure.

Likewise, one can gain access to the `pci_driver` structure from its embedded `device_driver` structure with:

```
struct device_driver *drv;
.....
struct pci_driver *pdrv = to_pci_drv (drv);
```

With a few exceptions (such as when doing **DMA** transfers) device drivers do not involve the generic structures and registration functions. For example, **PCI** devices fill in the `pci_driver` structure, and call `pci_register_driver()` (and some other functions) in order to get plugged into the system. However, these functions are written in terms of the underlying device model.

We have peeked at how **PCI** devices hook into the unified device model; we could do the same for other kinds of devices, such as **USB** and **network** and we would find the same kind of structural relations and embedded structures. Adding a new kind of device is just a question of following along the same path.

15.4 sysfs

Support for the **sysfs** virtual filesystem is built into all 2.6 kernels, and it should be mounted under `/sys`. However, the unified device model does not require mounting **sysfs** in order to function.

Let's take a look at what can be found using the 2.6.30 kernel; we warn you the exact layout of this filesystem has a tendency to mutate. Doing a top level directory command yields:

```
$ ls -F /sys
block/ bus/ class/ devices/ firmware/ fs/ kernel/ module/ power/
```

which displays the basic device hierarchy. The device model **sysfs** implementation also includes information not strictly related to hardware.

Network devices can be examined with:

```
$ ls -lF /sys/class/net
total 0
drwxr-xr-x 4 root root 0 Jul 28 01:33 eth0/
drwxr-xr-x 4 root root 0 Jul 28 01:33 eth1/
drwxr-xr-x 4 root root 0 Jul 28 01:33 lo/
```

and looking at the first Ethernet card gives:

```
$ ls -l /sys/class/net/eth0/
total 0
-r--r--r-- 1 root root 4096 Jul 28 01:33 address
```

```
-r--r--r-- 1 root root 4096 Jul 28 11:19 addr_len
-r--r--r-- 1 root root 4096 Jul 28 06:34 broadcast
-r--r--r-- 1 root root 4096 Jul 28 11:19 carrier
lrwxrwxrwx 1 root root    0 Jul 28 01:33 device ->
                             ../../../devices/pci0000:00/0000:00:1e.0/0000:05:02.0/
-r--r--r-- 1 root root 4096 Jul 28 11:19 dev_id
-r--r--r-- 1 root root 4096 Jul 28 11:19 dormant
-r--r--r-- 1 root root 4096 Jul 28 11:19 features
-rw-r--r-- 1 root root 4096 Jul 28 06:34 flags
-rw-r--r-- 1 root root 4096 Jul 28 11:19 ifalias
-r--r--r-- 1 root root 4096 Jul 28 06:34 ifindex
-r--r--r-- 1 root root 4096 Jul 28 11:19 iflink
-r--r--r-- 1 root root 4096 Jul 28 11:19 link_mode
-rw-r--r-- 1 root root 4096 Jul 28 11:19 mtu
-r--r--r-- 1 root root 4096 Jul 28 11:19 operstate
drwxr-xr-x 2 root root    0 Jul 28 11:19 power/
drwxr-xr-x 2 root root    0 Jul 28 06:35 statistics/
lrwxrwxrwx 1 root root    0 Jul 28 06:34 subsystem -> ../../net/
-rw-r--r-- 1 root root 4096 Jul 28 11:19 tx_queue_len
-r--r--r-- 1 root root 4096 Jul 28 01:33 type
-rw-r--r-- 1 root root 4096 Jul 28 01:33 uevent
```

Notice that typing out the simple entries just reads out values:

```
$ cat /sys/class/net/eth0/mtu
1500
```

in the way we are accustomed to getting information from the `/proc` filesystem. The intention with **sysfs** is to have one text value per line, although this is not expected to be rigorously enforced.

The underlying device and driver for the first network interface can be traced through the `device` and (the to be seen shortly) `driver` symbolic links. The directory for the first **PCI** bus looks like:

```
$ ls -F /sys/devices/pci0000:00

0000:00:00.0/  0000:00:1a.2/  0000:00:1c.4/  0000:00:1d.2/  0000:00:1f.2/     power/
0000:00:01.0/  0000:00:1a.7/  0000:00:1c.5/  0000:00:1d.7/  0000:00:1f.3/     uevent
0000:00:1a.0/  0000:00:1b.0/  0000:00:1d.0/  0000:00:1e.0/  firmware_node@
0000:00:1a.1/  0000:00:1c.0/  0000:00:1d.1/  0000:00:1f.0/  pci_bus:0000:00@
```

There is a subdirectory for each device, with the name giving the bus, device and function numbers; e.g., `0000:00:0a.0` means the first bus (0), eleventh device (10), and first function (0) on the device. Looking at the directory corresponding to the Ethernet card we see:

```
$ ls -l /sys/devices/pci0000:00/0000:00:1c.0
total 0
-rw-r--r-- 1 root root   4096 Oct 15 13:27 broken_parity_status
-r--r--r-- 1 root root   4096 Oct 15 13:27 class
-rw-r--r-- 1 root root    256 Oct 15 13:27 config
-r--r--r-- 1 root root   4096 Oct 15 13:27 device
lrwxrwxrwx 1 root root      0 Oct 15 13:27 driver -> ../../../bus/pci/drivers/skge
-rw------- 1 root root   4096 Oct 15 13:27 enable
-r--r--r-- 1 root root   4096 Oct 15 13:27 irq
-r--r--r-- 1 root root   4096 Oct 15 13:27 local_cpus
-r--r--r-- 1 root root   4096 Oct 15 13:27 modalias
-rw-r--r-- 1 root root   4096 Oct 15 13:27 msi_bus
drwxr-xr-x 3 root root      0 Oct 15 09:39 net
drwxr-xr-x 2 root root      0 Oct 15 13:27 power
-r--r--r-- 1 root root   4096 Oct 15 13:27 resource
-rw------- 1 root root  16384 Oct 15 13:27 resource0
-rw------- 1 root root    256 Oct 15 13:27 resource1
-r-------- 1 root root 131072 Oct 15 13:27 rom
lrwxrwxrwx 1 root root      0 Oct 15 13:27 subsystem -> ../../../bus/pci
-r--r--r-- 1 root root   4096 Oct 15 13:27 subsystem_device
-r--r--r-- 1 root root   4096 Oct 15 13:27 subsystem_vendor
-rw-r--r-- 1 root root   4096 Oct 15 13:27 uevent
-r--r--r-- 1 root root   4096 Oct 15 13:27 vendor
```


To see the full spectrum of information that is available with **sysfs** you'll just have to examine it.

15.5 Labs

Lab 1: Using libsysfs and sysfsutils.

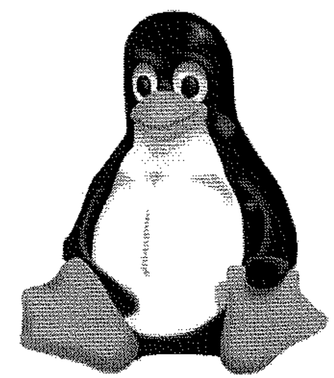
The **systool** multipurpose utility gives an easy interface for examining the `/sys` device tree, and is part of the **sysfstools** package. Currently it uses **libsysfs**, which is being deprecated in favor of **libhal**.

Do **man systool** and run the **systool** command without arguments. It should portray all bus types, devices classes, and root devices. Do `systool -h` to see how to use some of the additional arguments and options.

Explore!

Chapter 16

Firmware



We'll discuss binary firmware and how to use it.

16.1 What is Firmware?

Firmware consists of instructions and data embedded in a hardware device that are necessary for its proper functioning. This binary information can be stored in **ROM**, in rewritable **EEPROM**, or on flash media. However, vendors often find it is cheaper and more flexible to have the firmware loaded by the operating system.

While there is a gray line between deploying operating system-loaded firmware and the use of binary blobs, it is usually clear for a given device which is the more appropriate description. Use of firmware does not cause tainting of the **Linux** kernel. However, there are **Linux** distributions which have problems distributing drivers with require binary firmware, or which don't distribute the binary firmware itself.

We'll leave aside here the question of where to obtain the firmware for a given device that requires it, as it varies from device to device. Sometimes one even has to cut it out of the device driver for another operating system, as there may be no other method.

Furthermore, the question of whether firmware should be shipped with the kernel itself, and if not where it should reside on the filesystem is somewhat contentious and has not completely settled down. In some cases it does indeed ship with the kernel, and if not the precise location may be distribution-dependent. For most **Linux** systems one can generally find non-kernel shipped firmware under **/lib/firmware**.

16.2 Loading Firmware

Loading of firmware into drivers can be done with:

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

struct firmware {
        size_t size;
        const u8 *data;
};
MODULE_FIRMWARE(filename);

int request_firmware(const struct firmware **fw, const char *filename, struct device *device);
void release_firmware(const struct firmware *fw);
```

where `filename` is the name of the firmware file to be loaded; on any recent **Linux** system it should be placed under **/lib/firmware**. Upon successful return, `request_firmware()` places `size` bytes in the `data` field of the `firmware` structure.

For real devices attached to a physical bus such as **PCI**, one can easily get a hook into the `device` structure from the relevant bus-associated structure, such as `pci_dev`. For a pseudo-device (for which firmware probably doesn't make sense anyway), one would at least have to register the device and fill in some parts of the `device` structure before trying to load/unload the firmware.

There is a strong user-space component to how the firmware gets loaded, and it will vary among distributions. See **/usr/src/linux/Documentation/firmware_class** for details.

16.3 Labs

Lab 1: Loading Firmware

Write a module that loads some firmware from the filesystem. It should print out the contents.

In order to do this you'll need to place a firmware file under **/lib/firmware**. You can use just a text file for the purpose of this demonstration.

Since this is a pseudo device, you will have to declare and initialize a `device` structure. Minimally you must set the `void (*release)(struct device *dev)` field in this structure, and call

```
int dev_set_name (struct device *dev, const char *fmt, ...);
```


to set the device name, which can be read out with:

```
const char *dev_name(const struct device *dev);
```

(You may want to see what happens if you neglect setting one of these quantities.)

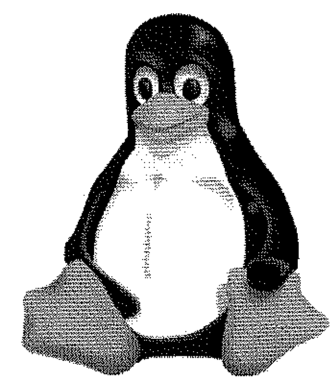
Make sure you call

```
device_register  (struct device *dev);
device_unregister(struct device *dev);
```

before requesting and after releasing the firmware.

Chapter 17

Memory Management and Allocation



We'll see how **Linux** distinguishes between virtual and physical memory and has them work together. We'll discuss the memory zone allocator scheme. We'll consider how memory is organized into pages and the various algorithms used to control and access them. We'll consider the various methods **Linux** uses to allocate memory within the kernel and device drivers, distinguishing between the **kmalloc()** and **vmalloc()** methods, and how to allocate whole pages or ranges of pages at once. We'll also consider how to grab larger amounts of memory at boot.

17.1 Virtual and Physical Memory

Linux uses a **virtual memory** system (**VM**), as do all modern operating systems: the virtual memory is **larger** than the physical memory.

Each process has its own, **protected** address space. Addresses are virtual and must be translated to and from physical addresses by the kernel whenever a process needs to access memory.

The kernel itself also uses virtual addresses; however the translation can be as simple as an offset depending on the architecture and the type of memory being used.

In the following diagram (for 32-bit platforms) the first 3 GB of virtual addresses are used for user-space memory and the upper GB is used for kernel-space memory (Other architectures have the same setup, but differing values for `PAGE_OFFSET`; for 64-bit platforms the value is in the stratosphere.)

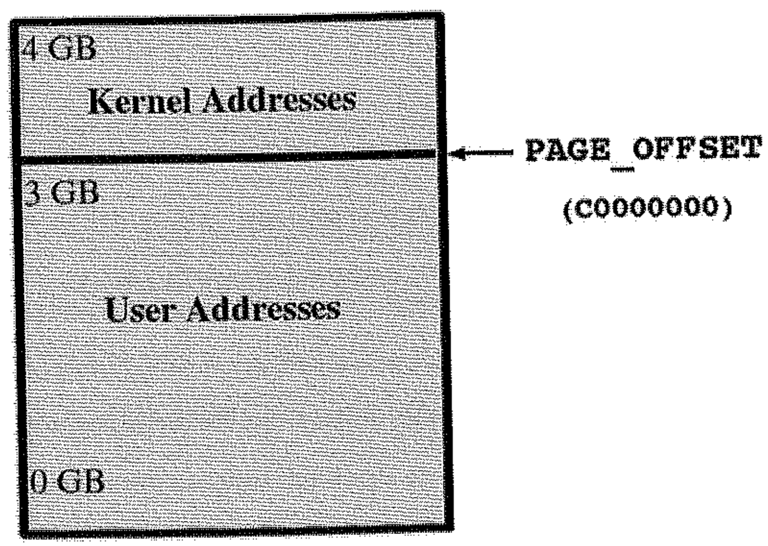


Figure 17.1: **User and kernel address regions**

The kernel allows fair shares of memory to be allocated to every running process, and coordinates when memory is shared among processes. In addition, **mapping** can be used to link a file directly to a process's virtual address space. Furthermore, certain areas of memory can be be protected against writing and/or code execution.

For a comprehensive review of **What Every Programmer Should Know About Memory**, see Ulrich Drepper's long article at **http://people.redhat.com/drepper/cpumemory.pdf**. This covers many issues in depth such as proper use of cache, alignment, **NUMA**, virtualization, etc.

17.2 Memory Zones

Linux uses a **zone allocator** memory algorithm, which is implemented in **/usr/src/linux/mm/page_alloc.c**. In this scheme, which has an object-oriented flavor, each zone has its own methods for basic memory operations, such as allocating and freeing pages of memory.

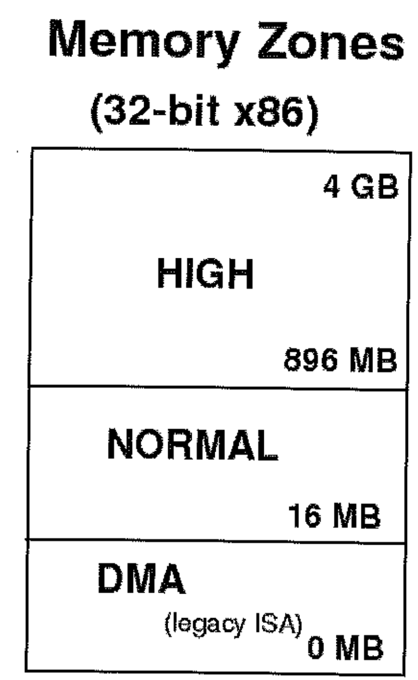


Figure 17.2: **DMA, normal and high memory**

There are three memory **zones**:

- **DMA**-capable memory must be used for **DMA** data transfers. Exactly what this means depends on the platform; for example, on **x86 ISA** devices it means the memory must lie below 16 MB.
- **High** memory requires special handling and has meaning only certain platforms. It allows access for up to 64 GB of physical memory. On the 32-bit **x86**, it means memory at and above 896 MB.
- **Normal** memory is everything else.

When memory is allocated the kernel examines what flags were associated with the request, and on that basis constructs a list of memory zones that can be used. When the flag `GFP_DMA` is requested, only pages in the **DMA** zone are considered. If `GFP_HIGHMEM` is specified all three zones can be used to get a free page. If neither of these flags are given, both the **normal** and **DMA** zones are considered. On platforms where high memory is not a concept, GFP_HIGHMEM has no effect; i.e., all memory is low memory, and memory is **flat**.

By looking at **/proc/zoneinfo** one can ascertain usage statistics for each operative zone.

While **Linux** permits up to 64 GB of memory to be used, the limit **per process** is a little less than 3 GB on 32-bit architectures. This is because there is only 4 GB of address space (i.e., it is 32-bit limited) and the topmost GB is reserved for kernel addresses. The little is somewhat less than 3 GB because of some address space being reserved for memory-mapped devices.

17.3 Page Tables

Memory is broken up into **pages** of fixed size (4 KB on **x86**). Portable code should never depend on a particular page size. To obtain the actual value kernel code can use the PAGE_SIZE macro and user-space programs can call the function getpagesize().

For 4K pages, the lower 12 bits of the virtual address contain the **offset**; the remaining bits contain the **virtual page frame number** (PFN).

Pages of virtual memory may be in any order in physical memory. The processor converts the virtual PFN into a physical one, using **page tables**. Each entry in the page table contains a **valid** flag, the PFN, and access control information.

If the requested virtual page is not **valid**, the kernel gets a **page fault** and then tries to get the proper page into physical memory. **Demand Paging** will try and bring the page in. The page may have been swapped out to disk.

If a page has been modified and there are insufficient free physical pages, a page is marked as **dirty** and will either be written to disk if the page corresponds to file-based data, or written to the **swap** file. Pages to be discarded are chosen with a **LRU** (Least Recently Used) algorithm.

Linux uses a **four-level Page Table**, even though the 32-bit **x86** processors have only two levels of page tables. (Before version 2.6.10, **Linux** used a three-level scheme.) This permits using the same functional methods for all architectures; traversing the superfluous levels involves fall-through functions.

If you use the 64 GB option on **x86**, **Linux** uses the **PAE** (Physical Address Extension) facility which gives an extra 4 bits of address space. In this it uses a true three-level scheme, rather than one in which one dimension is collapsed.

17.4 kmalloc()

The most common functions for allocating and freeing memory in the **Linux** kernel are:

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

void *kmalloc (unsigned int len, gfp_t gfp_mask);
void kfree (void *ptr);
```

Possible values for the gfp_mask argument are detailed in **/usr/src/linux/include/linux/gfp.h** and can be:

Table 17.1: **GFP memory allocation flags**

Value	Meaning
GFP_KERNEL	Block and cause going to sleep if the memory is not immediately available, allowing preemption to occur. This is the normal way of calling kmalloc().
GFP_ATOMIC	Return immediately if no pages are available. For instance, this might be done when kmalloc() is being called from an interrupt, where sleep would prevent receipt of other interrupts.
GFP_DMA	For buffers to be used with **ISA DMA** devices; is OR'ed with GFP_KERNEL or GFP_ATOMIC. Ensures the memory will be contiguous and falls under MAX_DMA_ADDRESS=16 MB on **x86** for **ISA** devices; for **PCI** this is unnecessary. The exact meaning of this flag is platform dependent.
GFP_USER	Used to allocate memory for a user . May sleep, and is a low priority request.
GFP_HIGHUSER	Like GFP_USER, but allocates from high memory
GFP_NOIO	Not to be used for filesystem calls, disallows I/O initiation.
GFP_NFS	For internal use.

Drivers normally use only the values GFP_KERNEL, GFP_ATOMIC, and GFP_DMA.

The in_interrupt() macro can be used to check if you are in interrupt or process context. For instance:

```
char *buffer = kmalloc (nbytes, in_interrupt() ? GFP_ATOMIC : GFP_KERNEL);
```

A similar macro, in_atomic(), also checks to see if you are in a preemptible context.

- **Note:** GFP_ATOMIC allocations are allowed to draw down memory resources more than those with GFP_KERNEL to lessen chances of failure; thus they should not be used when they are not necessary.

To allocate cleared memory, use

```
void *kzalloc (size_t size, gfp_t flags);
```

which calls kmalloc(size, flags) and then clears the allocated memory region.

One can also resize a dynamically allocated region with:

```
void *krealloc (const void *p, size_t new_size, gfp_t flags);
```

`kmalloc()` will return memory chunks in whatever power of 2 that matches or exceeds `len`. It doesn't clear memory, and will return `NULL` on failure, or a pointer to the allocated memory on success. The largest allocation that can be obtained is 1024 pages (4 MB on **x86**). For somewhat larger requests (more than a few KB) it is better to use the `__get_free_page()` functions.

17.5 __get_free_pages()

To allocate (and free) entire pages (or multiple pages) of memory in one fell swoop one can use:

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

unsigned long get_zeroed_page  (gfp_t gfp_mask);
unsigned long __get_free_page  (gfp_t gfp_mask);
unsigned long __get_free_pages (gfp_t gfp_mask, unsigned long order);

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

The `gfp_mask` argument is used in the same fashion as in `kmalloc()`.

`order` gives the number of pages (as a power of 2). The limit is 1024 pages, or `order = 10` (4 MB on **x86**). There is a function called `get_order()` defined in **/usr/src/linux/include/asm-generic/page.h** which can be used to determine the order given a number of bytes.

The `__get_free_pages()` function returns a pointer to the first byte of a memory area that is several pages long, and doesn't zero the area.

The `__get_free_page()` function doesn't clear the page; it is preferred over `get_zeroed_page()` because clearing the page might take longer than simply getting it.

It is important to free pages when they are no longer needed to avoid kernel memory leaks.

Example:

```
tty->read_buf = (unsigned char *) __get_free_page(
                              (in_interrupt()) ? GFP_ATOMIC : GFP_KERNEL);
if (!tty->read_buf)
          return -ENOMEM;
```


17.6 vmalloc()

`vmalloc()` allocates a contiguous memory region in the **virtual** address space:

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

void *vmalloc (unsigned long size);
void vfree (void *ptr);
```

`vmalloc()` can't be used when the real physical address is needed (such as for DMA), and can't be used at interrupt time; internally it uses `kmalloc()` with `GFP_KERNEL`.

While the allocated pages may not be consecutive in physical memory, the kernel sees them as a contiguous range of addresses. The resulting virtual addresses are higher than the top of physical memory.

More overhead is required than for `__get_free_pages()`, so this method shouldn't be used for small requests. In principle, `vmalloc()` can return up to the amount of physical RAM, but in reality one may obtain far less, depending on the platform and the amount of physical memory.

Example:

```
in_buf[dev]=(struct mbuf *)vmalloc(sizeof(struct mbuf));

if (in_buf[dev] == NULL)
 {
   printk(KERN_WARNING "Can't allocate buffer in_buf\n");
   my_devs[dev]->close(dev);
   return -EIO;
 }
```

Current `vmalloc()` allocations are exposed through **/proc/vmallocinfo**.

17.7 Early Allocations and bootmem()

The maximum amount of contiguous memory you can obtain through the various `__get_free_page()` functions is 1024 pages (4 MB on **x86**.) If you want more you can not do it from a module, but the kernel does offer some functions for doing this during boot:

```
#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 with **_pages** in their name allocate whole pages; the others are not page-aligned. The functions with **_low** make sure the memory locations obtained lie below `MAX_DMA_ADDRESS`. Otherwise, the memory allocation will be above that value.

It is impossible to free any memory allocated using these functions. However, once you have grabbed this large memory chunk you are free to run your own kind of memory allocator to reuse the memory as needed.

These functions are primarily intended for critical data structures that are allocated early in the boot process and are required throughout the life of the system. However, they can be used for other purposes.

17.8 Slabs and Cache Allocations

Suppose you need to allocate memory for an object that is less than a page in size, or is not a multiple of a page size and you don't want to waste space by requesting whole pages. Or suppose you need to create and destroy objects of the same size repeatedly, perhaps data structures or data buffers. These may be page size multiples or not.

In either case it would be very wasteful for the kernel to continually create and destroy these objects if they are going to be reused, and it is additionally wasteful to induce the kind of fragmentation that results from continually requesting partial pages.

You could allocate your own pool of memory and set up your own caching system, but **Linux** already has a well defined interface for doing this, and it should be used. **Linux** uses an algorithm based on the well-known **slab allocator** scheme. As part of this scheme you can create a special memory pool, or **cache** and add and remove objects from it (all of the same size) as needs require.

The kernel can dynamically shrink the cache if it has memory needs elsewhere, but it will not have to re-allocate a new object every time you need one if there are still wholly or partially unused **slabs** in the cache. Note that more than one **object** can be in a **slab**, whose size is going to be an integral number of pages.

The following functions create, set up, and destroy your own memory cache:

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

struct kmem_cache *kmem_cache_create (
   const char *name, size_t size,
   size_t offset, unsigned long flags,
   void (*ctor)(void *, struct kmem_cache *, unsigned long flags),
}
int kmem_cache_shrink  (struct kmem_cache *cache);
void kmem_cache_destroy (struct kmem_cache *cache);
```

These create a new memory cache of type `struct kmem_cache`, with the `name` argument serving to identify it. All objects in the cache (there can be any number) are `size` bytes in length, which cannot be more than 1024 pages (4 MB on **x86**).

The `offset` argument indicates alignment, or offset into the page for the objects you are allocating; normally you'll just give 0.

The last argument to `kmem_cache_create()` points to an optional **constructor** function used to initialize any objects before they are used; the header file contains more detailed information about the arguments and flags that can be passed to this rarely used function.

The `flags` argument is a bitmask of choices given in **/usr/src/linux/include/linux/slab.h**; the main ones are:

Table 17.3: **Memory cache flags**

Flag	Meaning
SLAB_HWCACHE_ALIGN	Force alignment of data objects on cache lines. This improves performance but may waste memory. Should be set for critical performance code.
SLAB_POISON	Fill the slab layer with the known value, a5a5a5a5. Good for catching access to uninitialized memory.
SLAB_RED_ZONE	Surround allocated memory with **red zones** that scream when touched, to detect buffer overruns.
SLAB_PANIC	Cause system panic upon allocation failure.
SLAB_DEBUG_FREE	Perform expensive checks on freeing objects
SLAB_CACHE_DMA	Make sure the allocation is in the **DMA** zone.

When your cache has been deployed, `name` will show up under **/proc/slabinfo**, and will show something like:

```
slabinfo - version: 1.1
kmem_cache           59     78    100    2    2    1
mycache               0      1   4096    0    1    1
ip_conntrack          0     11    352    0    1    1
tcp_tw_bucket         0      0     96    0    0    1
tcp_bind_bucket      12    113     32    1    1    1
.....
size-8192(DMA)        0      0   8192    0    0    2
size-8192             0      1   8192    0    1    2
....
size-32(DMA)          0      0     32    0    0    1
size-32             888   8814     32   69   78    1
```

where the meanings of the fields are:

- Cache name
- Number of active objects
- Total objects
- Object size
- Number of active slabs
- Total slabs
- Number of pages per slab

A dynamic and interactive view of the various caches on the system can be obtained by using the **slabtop** utility, where the elements can be sorted in many ways. One can see the same information by using the command `vmstat -m`.

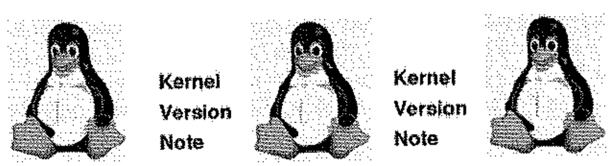
Now that you have created your memory cache, you can make any number of objects associated with it, and free them, with the functions:

```
void *kmem_cache_alloc( struct kmem_cache *cache, gfp_t gfp_mask);
void kmem_cache_free  ( struct kmem_cache *cache, void *);
```

pointing to the cache you have created as the first argument. The `gfp_mask` argument is the same as for `__get_free_pages()`. (If the memory doesn't already exist in the cache, it will be created using these flags.) The second argument to `kmem_cache_free()` simply points to what you got from `kmem_cache_alloc()`.

You can use the function `kmem_cache_shrink()` to release unused objects. When you no longer need your memory cache you must free it up with `kmem_cache_destroy()` (which shrinks the cache first); otherwise resources will not be freed. This function will fail if any object allocated to the cache has not been released.

Note it is also possible to set up a memory cache that never drops below a certain size using a **memory pool**, for which the **API** can be found in **/usr/src/linux/include/linux/mempool.h**. Such memory is taken outside of the normal memory management system and should be used only for critical purposes.



- The 2.6.22 kernel introduced the **SLUB** allocator as a drop-in replacement for the older **SLAB** implementation. Which one to use is a compile time option; in the 2.6.23 kernel **SLUB** was made the default.
- The new allocator has less of the complexity that evolved in the old one, has a smaller memory footprint, some performance enhancements, and easier debugging capabilities.
- Eventually **SLAB** will disappear, but due to the importance of having a stable cache allocator, this will only happen after **SLUB** has withstood the test of large scale deployment.

17.9 Labs

Lab 1: Memory Caches

Extend your character driver to allocate the driver's internal buffer by using your own memory cache. Make sure you free any slabs you create.

For extra credit create more than one object (perhaps every time you do a read or write) and make sure you release them all before destroying the cache.

Lab 2: Testing Maximum Memory Allocation

See how much memory you can obtain dynamically, using both `kmalloc()` and `__get_free_pages()`.

Start with requesting 1 page of memory, and then keep doubling until your request fails for each type fails.

Make sure you free any memory you receive.

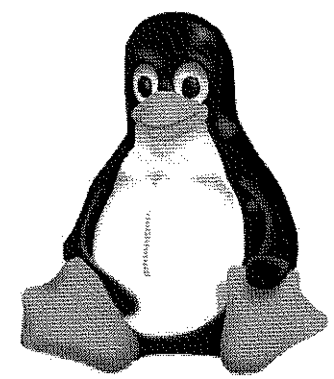
You'll probably want to use `GFP_ATOMIC` rather than `GFP_KERNEL`. (Why?)

If you have trouble getting enough memory due to memory fragmentation trying writing a poor-man's de-fragmenter, and then running again. The de-fragmenter can just be an application that grabs all available memory, uses it, and then releases it when done, thereby clearing the caches. You can also try the command `sync; echo 3 > /proc/sys/vm/drop_caches` .

Try the same thing with `vmalloc()`. Rather than doubling allocations, start at 4 MB and increase in 4 MB increments until failure results. Note this may hang while loading. (Why?)

Chapter 18

Transferring Between User and Kernel Space



We'll see how **Linux** handles the transfer of data between user and kernel space. We'll discuss the various functions used to accomplish this. We'll consider direct **kernel I/O**, which can be used to pin memory and enhance I/O throughput. We'll discuss **memory mapping**, explain the user-space system calls involved, and then examine the entry point into a character driver. We'll also consider the use of **Relay Channels**. We'll also show how to access files from within the Kernel.

18.1 Transferring Between Spaces

User-space applications work in a different (virtual) memory space than does the kernel.

When an address is passed to the kernel, it is the virtual address in user-space. An example would be the pointer to `buf` in the `read()` and `write()` driver entry points.

Any attempt from within the kernel to directly access these virtual pointers is a good recipe for disaster. As a matter of principle, these addresses may not be meaningful in kernel-space.

One might indeed get away with dereferencing a pointer passed from user-space - ***for a while.*** If a page gets swapped out, disaster will occur. The moral of the story is that one should **never** directly dereference a user-space pointer in kernel-space.

The functions which accomplish the transfers do two distinct things:

- Verify the user-space address, and handle any page faults that may occur if the page is currently not resident in memory.
- Perform a copy between the user and kernel addresses.

Using **raw** I/O or **memory mapping** can avoid copying.

18.2 put(get)_user() and copy_to(from)_user()

All the following functions can be used only in the context of a process, since they must refer to the current process's `task_struct` data structure in order to do the address translation. Calling them from an interrupt routine is another good recipe for disaster.

One should never surround the following transfer functions with a spinlock, as they may go to sleep, in which case your driver (or even the system) could get hung, as the spinlock might never be released.

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

access_ok (int type, unsigned long addr, unsigned long size);

int get_user (lvalue, ptr);
int __get_user (lvalue, ptr);

int put_user (expr, ptr);
int __put_user (expr, ptr);

unsigned long [__]copy_from_user (
        unsigned long to,
        unsigned long from,
        unsigned long len) ;

unsigned long [__]copy_to_user (
        unsigned long to,
        unsigned long from,
        unsigned long len) ;

long [__]strncpy_from_user (char *dst, const char *src, long count);
long  strlen_user (const char *str);
long strnlen_user (const char *str, long n);
unsigned long [__]clear_user (void *mem, unsigned long len);
```


- These functions are the **only** place in the kernel where page faults are resolved as they are in user-space, by demand paging or segmentation faults according to whether or not they are legal.
- This occurs only on pages for the user-space pointer; the kernel never swaps out pages for its own use and always allocates them with urgency and thus never has demand faulting for kernel memory.

access_ok()

`type` is `VERIFY_READ` or `VERIFY_WRITE` depending on what you want to do in *user-space*. For both use `VERIFY_WRITE`

`addr` is the address to be checked.

`size` is a byte count.

Is called by the most of the following functions; thus rarely needs to be called directly.

Returns `1` (true) if current process is allowed access; `0` on failure.

get_user()

Transfers data from **user** space to **kernel** space.

Assigns to `lvalue` data retrieved from the pointer `ptr`.

Is implemented as a macro, which depends on the type of `ptr`.

Calls `access_ok()` internally.

Retrieves a single value.

Returns `0` for success, `-EFAULT` otherwise.

__get_user()

Same as `get_user()` but doesn't call `access_ok()`. Use when safety is already assured.

put_user(), __put_user()

Transfers data from **kernel** space to **user** space.

Same as the `get_user()` functions, except the direction is reversed; Writes `expr` data to user space at `ptr`.

copy_from_user()

Transfers `len` bytes from **user** space to **kernel** space.

Calls `access_ok()` internally.

Also `__copy_from_user(to, from, len)`.

Returns the number of bytes ***not*** transferred. In error, the driver should return `-EFAULT`

copy_to_user()

Transfers `len` bytes from **kernel** space to **user** space.

Calls `access_ok()` internally.

Also `__copy_to_user(to, from, len)`.

Returns the number of bytes ***not*** transferred. In error, the driver should return `-EFAULT`

strncpy_from_user(), strlen_user(), clear_user()

The string functions work just as their names suggest, except the pointer of the string is in user-space.

The `clear_user()` function clears the contents of the memory location pointed to.

18.3 Direct transfer - Kernel I/O and Memory Mapping

There are two complementary methods **Linux** can use to avoid using the heretofore described transfer functions.

In the **kio** method, the kernel is given direct access to user-space memory pointers. The memory is locked down while the transfer goes on, making sure no pages swap out and the pointers remain valid. When the transfer is over the pinning is released. This is the basis of the **raw I/O** implementation. Note that if a file is opened with the non-standard `O_DIRECT` flag, **kio** will be used on that file.

In the **memory mapping** method, user-space is given direct access to kernel memory buffers, which may be memory regions residing directly on the device. The `mmap()` call is a standard POSIX system call.

Both methods avoid any buffering or caching for the data being transferred. They require longer to set up and shut down than the copying methods previously discussed. What is the best method depends on a number of factors, such as the size of the transfers, their frequency, the likelihood the data will be reused, etc.

18.4 Kernel I/O

Sometimes it is desirable to bypass the buffer and page caches entirely, and have I/O operations pass directly through to the device in raw form. This eliminates at least one copy operation. Large data base applications are often users of so-called **raw** I/O operations.

This facility can be used to lock down user-space buffers and use them directly in the kernel, without use of the `copy_to_user()`, `copy_from_user()` and related functions.

From user-space, one can force the kernel to use this kind of direct I/O, by opening a file with the `O_DIRECT` flag. This is a **gnu** extension, so you will also define the macro `_GNU_SOURCE`; i.e, you'll need something like:

```
#define _GNU_SOURCE
...
fd = open (filename, O_DIRECT | O_RDWR | O_CREAT | O_TRUNC, 0666);
```

Whenever this file descriptor is used, kernel I/O will be used.

Here's an example of a short program which copies a file, using direct I/O on the output file. We use `posix_memalign()` (or the older `memalign()`) instead of ordinary `malloc()` to ensure sector alignment. (All transfers must be sector aligned and an integral number of sectors long.)

```
/*
 args: 1 = input file, 2 = output file, [3 = chunk size]

 usage: %s infile ofile
        %s infile ofile 512
 */

#define _GNU_SOURCE
#define SECTOR_SIZE 512

#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#include <string.h>

int main (int argc, char *argv[])
{
```

```
    char *buf;
    int fdin, fdout, rcr, rcw;
/* default chunk = 1 page */
    int size = getpagesize ();
    if (argc > 3)
        size = atoi (argv[3]);

    /* open input file, must exist */
    fdin = open (argv[1], O_RDONLY);

    /* open output file, create or truncate */
    fdout = open (argv[2], O_DIRECT | O_RDWR | O_CREAT | O_TRUNC, 0666);

    /* use sector aligned memory region */
    /* buf = (char *)memalign (SECTOR_SIZE, size); */
    posix_memalign (&buf, SECTOR_SIZE, size);

    while ((rcr = read (fdin, buf, size)) > 0) {
        rcw = write (fdout, buf, rcr);
        printf ("in = %d, out = %d\n", rcr, rcw);
        if (rcr != rcw)
            printf ("Oops, BAD values -- not sector aligned perhaps\n");
    }
    close (fdin);
    close (fdout);
    exit (0);
}
```

18.5 Mapping User Pages

The get_user_pages() function provides a method of exposing user-space memory directly to the kernel, which can help avoid an extra copy; in some sense it is the inverse operation to **memory mapping** which makes kernel memory directly visible to the user side.

The essential function is:

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

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);
```

The first two arguments are the process and user address space involved; usually they are just `current` and `current->mm`.

The `start` argument gives the starting address of the user-space buffer of length `len` pages (not bytes). The `write` flag should be set if one desires to alter the buffer, and the `force` flag can be set to force access no matter what current permissions are.

The return value of this function is the number of pages mapped, and the pages argument receives an array to pointers for page structures. The final argument will be filled with an array of pointers to the `vm_area` structure containing each page, unless it is passed as `NULL`.

A typical use of this function might look like:

```
down_read (&current->mm->mmap_sem);
rc = get_user_pages (current, current->mm, (unsigned long) buf, npages, 1, 0, pages, NULL);
up_read (&current->mm->mmap_sem);
```

where a read lock is placed around the user-space memory region while the access is obtained.

One important thing to keep in mind is that one obtains only the pointer to the `struct page` that contains the user address. To get a useful kernel address for the page one has to use macros and functions such as:

```
#include <linux/pagemap.h>
....
char *kbuf = page_address (pages[i]);
```

or do

```
char *kbuf = kmap (pages[i]);
....
kunmap(pages[i]);
```

The second form also handles the case of high memory, but one has to be sure to do the unmapping when done. (In interrupt handlers, one needs to use the functions `kmap_atomic()`, `kunmap_atomic()`, but that can't happen when using `get_user_pages()` since you must be in process context anyway.)

Note that unless the buffer happens to be **page-aligned**, one only knows that the user address lies **somewhere** in the page; the offset is not furnished. One can produce page-aligned user memory with functions such as `posix_memalign()`, or use utilities which are alignment aware, such as **dd**. Programs such as **cat** are not.

It is also necessary to cleanup after any modification of the user pages; otherwise corruption may ensue as the virtual memory system has been bypassed. This means marking modified pages as **dirty** and releasing them from the page cache:

```
lock_page (pages[i]);
set_page_dirty (pages[i]);
unlock_page (pages[i]);
page_cache_release (pages[i]);
```

18.6 Memory Mapping

When a file is **memory mapped** the file (or part of it) can be associated with a range of linear addresses. Input and output operations on the file can be accomplished with simple memory references, rather than explicit I/O operations.

This can also be done on device nodes for direct access to hardware devices; in this case the driver must register and implement a proper `mmap()` entry point.

This method is not useful for stream-oriented devices. The mapped area must be a multiple of `PAGE_SIZE` in extent, and start on a page boundary.

Two basic kinds of memory mapping exist:

- In a **shared** memory map any operation on the memory region is completely equivalent to changing the file it represents. Changes are committed to disk (with the usual delays) and any process accessing the file or mapping it will see the changes.
- In a **private** memory map any changes are not committed to disk and are not seen by any other process. This is more efficient, but by design is used for a read only situation, or when the final saving of data is to be done by writing to another file.

Memory mapping can be more efficient than normal disk access, particularly when files are being shared by multiple processes, each one of whom can share access to certain pages, thereby minimizing memory usage and speeding access times.

18.7 User-Space Functions for mmap()

From the user side, memory mapping is done with:

```
#include <unistd.h>
#include <sys/mman.h>

void *mmap (void  *start,  size_t length, int prot, int flags, int fd, off_t offset);
int munmap (void *start, size_t length);
```

This requests the mapping into memory of `length` bytes, starting at offset `offset`, from the file specified by `fd`. The offset must be an integral number of pages.

The address `start` is a preferred address to map to; if 0 is given (the usual case), `mmap()` will choose the address and put it in the return value.

`prot` is the desired memory protection. It has bits:

Table 18.2: **mmap() memory protection bits**

Value	Meaning
PROT_EXEC	Pages may be executed.
PROT_READ	Pages may be read.
PROT_WRITE	Pages may be written.
PROT_NONE	Pages may not be accessed.

`flags` specifies the type of mapped object. It has bits:

Table 18.3: **mmap() flags**

Value	Meaning
MAP_FIXED	If `start` can't be used, fail.
MAP_SHARED	Share the mapping with all other processes.
MAP_PRIVATE	Create a private copy-on-write mapping.

Either `MAP_SHARED` or `MAP_PRIVATE` must be specified. Remember, a **private** mapping does not change the file on disk. Whatever changes are made will be lost when the process terminates.

Other non-POSIX flags can be specified (see **man mmap**.) In particular, the `MAP_ANONYMOUS` flag permits a mapping only in memory, without a file association.

Here's a simple example of using anonymous memory mapping to share memory between a parent and child:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/wait.h>

int main (int argc, char **argv)
{
    int fd = -1, size = 4096, status;
    char *area;
    pid_t pid;

    area =
        mmap (NULL, size, PROT_READ | PROT_WRITE,
              MAP_SHARED | MAP_ANONYMOUS, fd, 0);

    pid = fork ();
    if (pid == 0) {             /* child */
        strcpy (area, "This is a message from the child");
        printf ("Child has written: %s\n", area);
        exit (EXIT_SUCCESS);
    }
    if (pid > 0) {              /* parent */
        wait (&status);
```

```
        printf ("Parent has read:   %s\n", area);
        exit (EXIT_SUCCESS);
    }
    exit (EXIT_FAILURE);
}
```

munmap() deletes the mappings and causes further references to addresses within the range to generate invalid memory references.

See man mmap for further information on error codes.

18.8 Driver Entry Point for mmap()

From the kernel side, the driver entry point looks like:

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

int (*mmap) (struct file *filp, struct vm_area_struct *vma);
```

The vma_area_struct data structure is defined in **/usr/src/linux/include/linux/mm.h** and contains the important information. The basic elements are:

```
struct vm_area_struct {
...
   unsigned long vm_start;  /* Our start address within vm_mm. */
   unsigned long vm_end;    /* The first byte after our end address within vm_mm. */
....
   pgprot_t vm_page_prot;   /* Access permissions of this VMA. */
   unsigned long vm_flags;  /* Flags, listed below. */
....
   /* Function pointers to deal with this struct. */
   struct vm_operations_struct * vm_ops;

   /* Information about our backing store: */
   unsigned long vm_pgoff;  /* Offset (within vm_file) in PAGE_SIZE
                               units, *not* PAGE_CACHE_SIZE */
....
};
```

The vm_ops structure can be used to override default operations. Pointers can be given for functions to: open(), close(), unmap(), protect(), sync(), advice(), swapout(), swapin() ....

A simple example serves to show how the fields are used:

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

int my_mmap (struct file *file, struct vm_area_struct *vma)
{
  if( remap_pfn_range (vma, vma->vm_start, vma->vm_pgoff,
      vma->vm_end-vma->vm_start, vma->vm_page_prot ))
```



```
        return -EAGAIN;
  return 0;
}
```

Most of the work is done by the function remap_pfn_range(). Note that this function does allow mapping memory above the 4 GB barrier.

Here is a simple example of a program to test the mmap() entry:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/mman.h>

#define DEATH(mess) { perror(mess); exit(errno); }

int main (int argc, char **argv)
{
    int fd, size, rc, j;
    char *area, *tmp, *nodename = "/dev/mycdrv";
    char c[2] = "CX";

    if (argc > 1)
        nodename = argv[1];

    size = getpagesize ();      /* use one page by default */
    if (argc > 2)
        size = atoi (argv[2]);

    printf (" Memory Mapping Node: %s, of size %d bytes\n", nodename, size);

    if ((fd = open (nodename, O_RDWR)) < 0)
        DEATH ("problems opening the node ");

    area = mmap (NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    if (area == MAP_FAILED)
        DEATH ("error mmaping");

    /* can close the file now */

    close (fd);

    /* put the string repeatedly in the file */

    tmp = area;
    for (j = 0; j < size - 1; j += 2, tmp += 2)
        memcpy (tmp, &c, 2);

    /* just cat out the file to see if it worked */

    rc = write (STDOUT_FILENO, area, size);
```

```
    if (rc != size)
        DEATH ("problems writing");

    exit (EXIT_SUCCESS);
}
```

Here is a simple driver with a mmap() entry point:

```
/* Sample Character Driver with mmap'ing */

#include <linux/module.h>       /* for modules */
#include <linux/fs.h>           /* file_operations */
#include <linux/uaccess.h>      /* copy_(to,from)_user */
#include <linux/init.h>         /* module_init, module_exit */
#include <linux/slab.h>         /* kmalloc */
#include <linux/cdev.h>         /* cdev utilities */
#include <linux/mm.h>           /* remap_pfn_range */

#define MYDEV_NAME "mycdrv"

static dev_t first;
static unsigned int count = 1;
static int my_major = 700, my_minor = 0;
static struct cdev *my_cdev;

static int mycdrv_mmap (struct file *file, struct vm_area_struct *vma)
{
    printk (KERN_INFO "I entered the mmap function\n");
    if (remap_pfn_range (vma, vma->vm_start,
                         vma->vm_pgoff,
                         vma->vm_end - vma->vm_start, vma->vm_page_prot)) {
        return -EAGAIN;
    }

    return 0;
}

/* don't bother with open, release, read and write */

static struct file_operations mycdrv_fops = {
    .owner = THIS_MODULE,
    .mmap = mycdrv_mmap,
};
static int __init my_init (void)
{
    first = MKDEV (my_major, my_minor);
    register_chrdev_region (first, count, MYDEV_NAME);
    my_cdev = cdev_alloc ();
    cdev_init (my_cdev, &mycdrv_fops);
    cdev_add (my_cdev, first, count);
    printk (KERN_INFO "\nSucceeded in registering character device %s\n",
            MYDEV_NAME);
    return 0;
```



```
}
static void __exit my_exit (void)
{
    cdev_del (my_cdev);
    unregister_chrdev_region (first, count);
    printk (KERN_INFO "\ndevice unregistered\n");
}

module_init (my_init);
module_exit (my_exit);

MODULE_AUTHOR ("Jerry Cooperstein");
MODULE_DESCRIPTION ("MODULE_DESCRIPTION_NAME");
MODULE_LICENSE ("GPL v2");
```

18.9 Relay Channels

One often comes up with the need to transfer information between kernel-space and user-space, but not all needs are the same. One may or may not need bi-directionality, efficiency, or promptness. Or one may be working with or without a device driver, and have large or small amounts of data.

The **Relay Channel** interface (formerly known as **relayfs**) provides a simple to use mechanism that works beautifully when the direction is one way: from kernel to user.

Kernel clients fill up **channel buffers** with no special constraints on the data form. Users get access to the data with normal system calls, generally **read()** and/or **mmap()**, exercised on data files (by default one for each CPU) that are treated much like normal pipes.

For each relay channel, there is one buffer per CPU. In turn, each buffer has one or more sub-buffers.

When a sub-buffer is too full to fit a new chunk of data, or message, the next buffer (if available) is used; messages are never split between sub-buffers (so a message should not be bigger than a sub-buffer.) User-space can be notified that a sub-buffer is full.

The buffer can be set up in either overwrite or no-overwrite mode (the default); in the second mode, kernel clients will block until readers empty the buffer.

When the user-space application accesses the data with `read()` calls, any padding at the end of sub-buffers is removed and the buffers are drained.

When user-space application accesses the data with `mmap()` calls, the entire buffer (including all sub-buffers) must be mapped and no draining occurs. This is more efficient than just using reads, but is also more complex.

Here's a complete list of the system calls that can be used on a relay channel:

- `open()`, `close()`: open and close an existing channel buffer. If no other process, or kernel client, is still using the buffer, the channel is freed upon closing.
- `read()`: Consume bytes from the channel. In no-overwrite mode it is fine if kernel clients are writing simultaneously, but in overwrite mode unpredictable outcomes can happen. Sub-buffer padding is not seen by readers.
- `mmap()`, `munmap()`: The entire buffer must be mapped and there is no draining.

- `sendfile()`: Drains like a read.
- `poll()`: User applications are notified when a sub-buffer boundary is reached, and the flags `POLLIN`, `POLLRDNORM`, `POLLERR` are supported.

While the work of this mechanism could be done using other methods, such as using the `/proc` filesystem, `ioctl()` commands on either real or pseudo devices, improper use of `printk()` statements, or worst, accessing a log file directly from the kernel, the use of `relay channels` offers a clean (and approved) method and should be considered strongly.

18.10 Relay API

Opening and closing a relay channel is done with

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

struct rchan *relay_open(const char *base_filename,
                         struct dentry *parent,
                         size_t subbuf_size,
                         size_t n_subbufs,
                         struct rchan_callbacks *cb
                         void *private_data);
void relay_close(struct rchan *chan);
```

which associates a file with the channel for each CPU; e.g., if `base_filename = "my_chan"`, the files will be named `my_chan0, my_chan1, my_chan2`. The associated files will appear in the directory pointed to by `parent`; if this is `NULL`, they be in the host filesystem's root directory.

Each of the `n_subbufs` sub-buffers is of size `subbuf_size`, so the total size of the buffer is `subbuf_size * n_subbufs`. Writes by kernel clients should not be bigger than `subbuf_size` since they can't be split across sub-buffers.

When one wants to write into a relay channel, it is done with:

```
void relay_write(struct rchan *chan, const void *data, size_t length);
```

and the information will appear in the associated pseudofile. The final argument to `relay_open()` is to a table of callback functions:

```
struct rchan_callbacks {
    int (*subbuf_start) (struct rchan_buf *buf,
                         void *subbuf,
                         void *prev_subbuf,
                         size_t prev_padding);
    void (*buf_mapped)  (struct rchan_buf *buf,
                         struct file *filp);
    void (*buf_unmapped)(struct rchan_buf *buf,
                         struct file *filp);
    struct dentry *(*create_buf_file)(const char *filename,
                         struct dentry *parent,
                         int mode,
                         struct rchan_buf *buf,
                         int *is_global);
    int (*remove_buf_file)(struct dentry *dentry);
}
```


`subbuf_start()` is called when one switches to a new sub-buffer. `buf_mapped(), buf_unmapped()` are called when the buffer is memory mapped or unmapped.

`create_buf_file(), remove_buf_file()` create (and remove) the files associated with the relay channel. Note that if the parameter `is_global` is not zero, there will be only one file even on multiple CPUs; in that case you will explicitly have to take care of any race conditions. The `mode` argument gives the usual permissions and `parent` is obviously the parent directory. `filename` has to be **created/removed** by this method.

There is no apriori requirement for where these files should go. A convenient place is the `debugfs` filesystem. In that case one could have:

```
static struct dentry
*create_buf_file_handler(const char *filename,
                         struct dentry *parent,
                         int mode,
                         struct rchan_buf *buf,
                         int *is_global)
{
    return debugfs_create_file(filename, mode, parent, buf,
                               &relay_file_operations);
}
static int remove_buf_file_handler(struct dentry *dentry)
{
    debugfs_remove(dentry);
    return 0;
}
static struct rchan_callbacks relay_callbacks =
{
        .create_buf_file = create_buf_file_handler,
        .remove_buf_file = remove_buf_file_handler,
};
```

where `relay_file_operations` is the `file_operations` structure defined in **/usr/src/linux /kernel/relay.c**.

There are additional callback and utility functions that can be used with relay channels, and one can take control at a lower level than we have indicated. Working with memory mapping requires a little more work than just using `read()` calls. However, we would recommend starting with what we have described before trying to master some of the intricacies, especially when working in overwrite mode.

18.11 Accessing Files from the Kernel

A perennial question is "How do I do file I/O from within the kernel?" This is a **bad idea**. It is full of problems involving, stability, race conditions, and security. For an excellent explanation of why this operation is really only suitable as a learning exercise, see **http://www.cs.helsinki.fi/linux/linux-kernel/2003-23/1447.html**.

You can't accomplish file I/O without a process context; the kernel has to borrow one or create one; borrowing is extremely dangerous as you may corrupt the context of the loaner; creating requires a new kernel thread.

For a similar method to what is given below, see the article by Greg Kroah-Hartman at **http://www.linuxjournal.com/article/8110**.

One must set the address space to a user one before dealing with files, and then reset it when done. The macros for handling this are:

```
get_ds();
get_fs();
set_fs(x);
```

The macro `set_fs(x)` sets which **data segment** to use, where `x` can be `KERNEL_DS` or `USER_DS`. The macro `get_ds()` is just a shorthand for `KERNEL_DS`. The full definitions can be found in **/usr/src/linux/arch/x86/include/asm/uaccess.h** on most architectures.

While kernel developers have made directly dealing with files deliberately difficult, however, there does exist a `kernel_read()` function that can be used, and we'll define a `kernel_write()` function below to go along with it.

Here's an example of how to do it:

Example:

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/uaccess.h>
#include <linux/slab.h>
#include <linux/fs.h>

static char *filename = "/tmp/tempfile";
module_param (filename, charp, S_IRUGO);

int kernel_write (struct file *file, unsigned long offset,
                  char *addr, unsigned long count)
{
    mm_segment_t old_fs;
    loff_t pos = offset;
    int result;

    old_fs = get_fs ();
    set_fs (get_ds ());
    /* The cast to a user pointer is valid due to the set_fs() */
    result = vfs_write (file, (void __user *)addr, count, &pos);
    set_fs (old_fs);
    return result;
}
```



```
#define NBYTES_TO_READ 20

/* adapted from kernel_read() in kernel/exec.c */

static int __init my_init (void)
{
    struct file *f;
    int nbytes, j;
    char *buffer;
    char newstring[] = "NEWSTRING";

    buffer = kmalloc (PAGE_SIZE, GFP_KERNEL);
    printk (KERN_INFO "Trying to open file = %s\n", filename);
    f = filp_open (filename, O_RDWR, S_IWUSR | S_IRUSR);

    if (IS_ERR (f)) {
        printk (KERN_INFO "error opening %s\n", filename);
        kfree (buffer);
        return -EIO;
    }

    nbytes = kernel_read (f, f->f_pos, buffer, NBYTES_TO_READ);

    printk (KERN_INFO "I read nbytes = %d, which were: \n\n", nbytes);
    for (j = 0; j < nbytes; j++)
        printk (KERN_INFO "%c", buffer[j]);

    strcpy (buffer, newstring);
    nbytes = kernel_write (f, f->f_pos, buffer, strlen (newstring) + 1);
    printk (KERN_INFO "\n\n I wrote nbytes = %d, which were %s \n", nbytes,
            newstring);

    filp_close (f, NULL);
    kfree (buffer);

    return 0;
}

static void __exit my_exit (void)
{
    printk (KERN_INFO "\nclosing up\n");
}

module_init (my_init);
module_exit (my_exit);
```

Such a method should never be used in code that is submitted to the kernel tree.

18.12 Labs

Lab 1: Using get_user() and put_user().

Adapt your character driver to use `get_user()` and `put_user()`.

Lab 2: Mapping User Pages

Use the character device driver, adapt it to use `get_user_pages()` for the `read()` and `write()` entry points.

To properly exercise this you'll need to use a page-aligned utility such as **dd**, or write page-aligned reading and writing programs.

Lab 3: Memory Mapping an Allocated Region

Write a character driver that implements a `mmap()` entry point that memory maps a kernel buffer, allocated dynamically (probably during initialization).

There should also be `read()` and `write()` entry points.

Optionally, you may want to use an `ioctl()` command to tell user-space the size of the kernel buffer being memory mapped.

Note: This is not an easy exercise to do properly, so if time is lacking you may merely experiment with the solutions.

Lab 4: Using Relay Channels.

Write a kernel module that opens up a relay channel and makes the associated files visible in the **debugfs** filesystem.

Make sure you mount the filesystem (if necessary) with

```
mount -t debugfs none /sys/kernel/debug
```

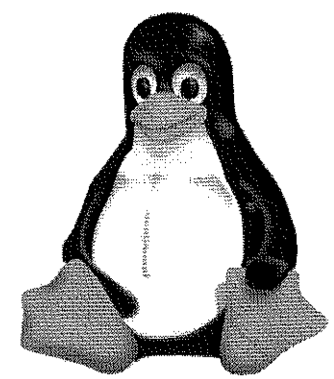
Have the initialization routine write a series of entries into the channel. While the kernel module is loaded, try reading from it using `read()` and `mmap()`.

If you read more than once on the open file descriptor what do you see?

For more advanced exercises, you might try making sure your kernel client writes over sub-buffer boundaries, or writes into the channel from other functions such as an interrupt routine, or other entry points.

Chapter 19

Sleeping and Wait Queues



We'll discuss **wait queues**. We'll consider how tasks can be put to sleep, and how they can be woken up. We'll also consider the **poll()** entry point, and methods of interrupt handling from user-space..

19.1 What are Wait Queues?

Wait queues are used when a task running in kernel mode has reached a condition where it needs to wait for some condition to be fulfilled. For instance it may need to wait for data to arrive on a peripheral device.

At such times it is necessary for the task to go to sleep until whatever condition or resource it is waiting for is ready. When the resource becomes available, or the condition becomes true, (perhaps signalled by the arrival of an interrupt) it will become necessary to **wake up** the sleeping task.

There can be many wait queues in the system and they are connected in a linked list. In addition more than one task can be placed on a given wait queue.

Another way to understand wait queues is to think of **task organization** and queues. There is a linked list of all tasks who have `TASK_RUNNING` in the state field of their `task_struct`, called the **runqueue**. A task which is scheduled out but would like to run as soon as a timeslice is available is **not sleeping**; it still has `TASK_RUNNING` as its state.

Sleeping tasks (those with a state of `TASK_INTERRUPTIBLE`, `TASK_UNINTERRUPTIBLE`, or `TASK_KILLABLE`) go instead into one of many possible wait queues, each of which corresponds to getting woken up by a particular event or class of events, at which point the sleeping task can go back to the runqueue.

The sleeping and waking up functions come in two forms, **interruptible** and **uninterruptible**. Uninterruptible sleep is not woken up by a signal and as such should be rarely used, especially in device drivers. It is quite difficult to get out of a task hung in this situation; short of a reboot one may be able to cause a wake up function to be called by terminating an ancestor process.

The `TASK_KILLABLE` state is woken up only a **fatal** signal (while `TASK_INTERRUPTIBLE` wakes up with **any** signal.) It was introduced in the 2.6.25 kernel.

When a wait queue is woken up, all tasks on the wait queue are roused (unless an **exclusive** sleep is used, as we shall see.)

It is very easy to hang a system with improper use of wait queues. In particular, kernel threads of execution such as interrupt service routines should **never** go to sleep.

The data structure used by wait queues is of the type `wait_queue_head_t`, usually just called a **wait queue**. It is explicitly declared and initialized with with the statements

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

wait_queue_head_t wq;
init_waitqueue_head (&wq);
```

If the wait queue is not allocated at run time it can be declared and initialized with the macro

```
DECLARE_WAIT_QUEUE_HEAD(wq);
```

Don't forget to initialize a wait queue.

19.2 Going to Sleep and Waking Up

Now that we have set up a wait queue, we need to use functions for putting a task to sleep and for waking it up. These are

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



```
wait_event                (wait_queue_head_t wq, int condition);
wait_event_interruptible  (wait_queue_head_t wq, int condition);
wait_event_killable       (wait_queue_head_t wq, int condition);

void wake_up                   (wait_queue_head_t *wq);
void wake_up_interruptible     (wait_queue_head_t *wq);
```

The `wait_event()` calls are actually macros, not functions. They take `wq`, not `*wq`, as their argument.

The proper wake up call should be paired with the originating sleep call. (However, `wait_event_killable()` should be paired with `wake_up()`, which isn't obvious.)

In general you will want to use the **interruptible** wait functions which return 0 if they return due to a wake up call and `-ERESTARTSYS` if they return due to a signal arriving. The other forms are not aborted by a signal and are only used by critical sections of the kernel, such as while waiting for a swap page to be read from disk.

When you use the interruptible forms, you'll always have to check upon awakening whether you woke up because a signal arrived, or there was an explicit wake up call. The `signal_pending(current)` macro can be used for this purpose.

The `condition` test has two important purposes:

- It helps avoid the race condition in which a task is designated to sleep but the wake up call arrives before the change in state is complete; the condition is checked before actually putting the task to sleep.
- The condition is checked upon waking up and if it is not true the task remains asleep. This helps avoid another class of race conditions where a task is put to sleep again before it has a chance to really wake up.

You will still have to call one of the **wake_up** functions when using these macros; they do not just set up a spinning `while` loop until the argument given in `condition` evaluates as true (non-zero).

Sometimes you want to ensure you don't sleep too long. For this purpose one can use:

```
wait_event_timeout                (wait_queue_head_t wq, int condition, long timeout);
wait_event_interruptible_timeout  (wait_queue_head_t wq, int condition, long timeout);
```

where the timeout is specified in `jiffies`. If the task returns upon timeout, these functions return 0. If they return earlier, they return the remaining `jiffies` in the timeout period. If the interruptible form returns due to a signal, it returns `-ERSTARTSYS`.

The waking functions will rouse **all** sleepers on the specified wait queue. There is no guarantee about the order in which they will be woken up; they will be scheduled in by priority algorithms rather than **FIFO** or **LIFO**. Furthermore, the tasks can be woken up on any CPU. A little more control can be obtained with the function:

```
void wake_up_interruptible_sync (wait_queue_head_t *wq);
```

which checks whether the task being woken up has a higher priority than the currently running one, and if so, invokes the scheduler if possible. However, this is rarely done.

Thus a simple use of wait queues would include a code fragment like:

```
#include <linux/sched.h>
DECLARE_WAIT_QUEUE_HEAD(wq)

static int fun1 ( ... )
{
    ...
    printk(KERN_INFO "task %i (%s) going to sleep\n", current->pid, current->comm);
    wait_event_interruptible(wq, dataready);
    printk(KERN_INFO "awoken %i (%s)\n", current->pid, current->comm);
    if (signal_pending (current))
          return -ERESTARTSYS;
    ...
    dataready = 0;
}
static int fun2 ( ... )
{
    ...
    printk(KERN_INFO "task %i (%s) awakening sleepers...\n", current->pid, current->comm);
          dataready = 1;
    wake_up_interruptible(&wq);
    ...
}
```

(Note the variable `dataready` should probably be an atomic one, or be protected by some kind of lock.)

19.3 Going to Sleep Details

Let's look in some detail at the code for entering a wait, or going to sleep. The macro `wait_event()` is defined in **/usr/src/linux/include/linux/wait.h**:

```
2.6.31: 196 #define wait_event(wq, condition)                          \
2.6.31: 197 do {                                                   \
2.6.31: 198      if (condition)                                    \
2.6.31: 199            break;                                      \
2.6.31: 200      __wait_event(wq, condition);                      \
2.6.31: 201 } while (0)
....
2.6.31: 171 #define __wait_event(wq, condition)                        \
2.6.31: 172 do {                                                   \
2.6.31: 173      DEFINE_WAIT(__wait);                              \
2.6.31: 174                                                        \
2.6.31: 175      for (;;) {                                        \
2.6.31: 176            prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE); \
2.6.31: 177            if (condition)                              \
2.6.31: 178                  break;                                \
2.6.31: 179            schedule();                                 \
2.6.31: 180      }                                                 \
2.6.31: 181      finish_wait(&wq, &__wait);                        \
2.6.31: 182 } while (0)
```


The first thing to do is to check if `condition` is true, and if so, avoid going to sleep at all. This avoids the race condition in which the condition is reset and a wake up call is issued after the task is requested to go to sleep but before it actually does so.

Then one enters the macro where the real work is done, `__wait_event()`, where the first thing to do is `DEFINE_WAIT (name)`, which is equivalent to:

```
wait_queue_t name;
init_wait (&name);
```

which creates and initializes the wait queue.

The next thing to do is to add the wait queue entry to the queue, and reset the state of the task, which is done by:

```
void prepare_to_wait (wait_queue_head_t *queue, wait queue_t *wait, int state);
```

Once again one checks `condition` to avoid a race condition, e.g., a missed wake up call, in which case the sleep is once again avoided. Assuming this condition is not true, one calls `schedule()` to schedule in another task; the current one can't be scheduled in because its state is `TASK_UNINTERRUPTIBLE`.

The next lines of code will only be entered after the state has been reset by a wake up call, and the task is again available for scheduling and has been granted a time slice. The `for()` loop makes sure the `condition` is really true, and if not continues sleep until it is.

When the sleep is truly finished, one calls:

```
void finish_wait (wait_queue_head_t *queue, wait_queue_t *wait);
```

which does whatever cleanup is needed.

The `wait_event_interruptible()` macro is almost the same except that it sets the state to `TASK_INTERRUPTIBLE` and the `for()` loop is replaced with:

```
2.6.31: 246      for (;;) {                                        \
2.6.31: 247            prepare_to_wait(&wq, &__wait, TASK_INTERRUPTIBLE);   \
2.6.31: 248            if (condition)                              \
2.6.31: 249                  break;                                \
2.6.31: 250            if (!signal_pending(current)) {             \
2.6.31: 251                  schedule();                           \
2.6.31: 252                  continue;                             \
2.6.31: 253            }                                           \
2.6.31: 254            ret = -ERESTARTSYS;                         \
2.6.31: 255            break;                                      \
2.6.31: 256      }                                                 \
2.6.31: 257      finish_wait(&wq, &__wait);                        \
2.6.31: 258 } while (0)
```

which checks to see if the sleep ended because of an incoming signal, and if so returns the value `-ERESTARTSYS`.

The **timeout** variations use for the `for()` loop:

```
2.6.31: 207    for (;;) {                                                   \
2.6.31: 208            prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE); \
2.6.31: 209            if (condition)                                       \
2.6.31: 210                    break;                                       \
2.6.31: 211            ret = schedule_timeout(ret);                         \
2.6.31: 212            if (!ret)                                            \
2.6.31: 213                    break;                                       \
2.6.31: 214    }                                                            \
```

in which `schedule_timeout()` causes the scheduler to get called if the timeout period elapses.

19.4 Exclusive Sleeping

So far we have dealt only with so-called **non-exclusive** sleeping tasks. For instance, a number of tasks may be waiting for termination of a disk operation, and once it has completed they will all need to wake up and resume.

If more than one task is waiting for **exclusive** access to a resource (one that only one can use at a time) then this kind of wake up is inefficient and leads to the **thundering herd** problem, where all sleepers are woken up even though only one of them can use the resource at a time.

In this case new functions are required. Setting up the wait involves the inline macro function:

```
wait_event_interruptible_exclusive (wait_queue_head_t wq, int condition);
```

(At this time, there is no non-interruptible equivalent convenience macro, but one can construct a non-interruptible sleep from lower level primitives.)

The usual wake up functions can be used; in this case only one sleeper will be woken up. If more control is needed a number of new wake up functions can be used:

```
void wake_up_all                 ( wait_queue_head_t *wq);
void wake_up_interruptible_all   ( wait_queue_head_t *wq);
void wake_up_nr                  ( wait_queue_head_t *wq, int nr)
void wake_up_sync_nr             ( wait_queue_head_t *wq, int nr)
void wake_up_interruptible_nr    ( wait_queue_head_t *wq, int nr)
void wake_up_interruptible_sync_nr ( wait_queue_head_t *wq, int nr)
```

The ones with `all` in the name wake up all tasks in the queue, just as in the non-exclusive case, but those with `_nr` awaken only nr tasks (typically nr=1.)

19.5 Waking Up Details

All the wake up calls are macros that invoke the basic `__wake_up()` call and are defined in **/usr/src/linux/include/linux/wait.h**:

```
2.6.31: 149 #define wake_up(x)                  __wake_up(x, TASK_NORMAL, 1, NULL)
2.6.31: 150 #define wake_up_nr(x, nr)           __wake_up(x, TASK_NORMAL, nr, NULL)
```



```
2.6.31: 151 #define wake_up_all(x)              __wake_up(x, TASK_NORMAL, 0, NULL)
2.6.31: 152 #define wake_up_locked(x)           __wake_up_locked((x), TASK_NORMAL)
2.6.31: 153
2.6.31: 154 #define wake_up_interruptible(x)    __wake_up(x, TASK_INTERRUPTIBLE, 1, NULL)
2.6.31: 155 #define wake_up_interruptible_nr(x, nr) __wake_up(x, TASK_INTERRUPTIBLE,
                nr, NULL)
2.6.31: 156 #define wake_up_interruptible_all(x)  __wake_up(x, TASK_INTERRUPTIBLE, 0,
                NULL)
2.6.31: 157 #define wake_up_interruptible_sync(x) __wake_up_sync((x),
                TASK_INTERRUPTIBLE, 1)
```

where

```
TASK_NORMAL = TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE
```

The code for the core `__wake_up()` function is in **/usr/src/linux/kernel/sched.c**:

```
2.6.31:5552 void __wake_up(wait_queue_head_t *q, unsigned int mode,
2.6.31:5553                         int nr_exclusive, void *key)
2.6.31:5554 {
2.6.31:5555     unsigned long flags;
2.6.31:5556
2.6.31:5557     spin_lock_irqsave(&q->lock, flags);
2.6.31:5558     __wake_up_common(q, mode, nr_exclusive, 0, key);
2.6.31:5559     spin_unlock_irqrestore(&q->lock, flags);
2.6.31:5560 }
2.6.31:5561 EXPORT_SYMBOL(__wake_up);
```

which takes out an interrupt blocking spinlock, and then passes the work off to `__wake_up_common()`:

```
2.6.31:5528 static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
2.6.31:5529                         int nr_exclusive, int sync, void *key)
2.6.31:5530 {
2.6.31:5531     wait_queue_t *curr, *next;
2.6.31:5532
2.6.31:5533     list_for_each_entry_safe(curr, next, &q->task_list, task_list) {
2.6.31:5534             unsigned flags = curr->flags;
2.6.31:5535
2.6.31:5536             if (curr->func(curr, mode, sync, key) &&
2.6.31:5537                             (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
2.6.31:5538                     break;
2.6.31:5539     }
2.6.31:5540 }
```

The function cycles through the linked list of wait queues, and for each task placed on a wait queue it calls the wake up function (`curr->func()`) which by default is set to be `default_wake_function()`. (The ability to use an alternative wake up function appeared in the 2.6 kernel) After doing so it checks to see whether or not it is an exclusive wait, and if so properly decrements the number of remaining tasks to be woken up.

The default wake up function in turn just calls `try_to_wake_up()`:

```
2.6.31:5512 int default_wake_function(wait_queue_t *curr, unsigned mode, int sync,
2.6.31:5513                           void *key)
2.6.31:5514 {
2.6.31:5515        return try_to_wake_up(curr->private, mode, sync);
2.6.31:5516 }
2.6.31:5517 EXPORT_SYMBOL(default_wake_function);
```

Now we actually do the wake up with

```
int try_to_wake_up (task_t * p, unsigned int state, int sync);
```

which is a long and complicated function, mostly because of the necessity of ensuring a task is not already running on another cpu. If not, it will set the state to `TASK_RUNNING`, and enable the task to be rescheduled.

19.6 Polling

Applications often keep their eye on a number of file descriptors to see whether or not it is possible to do I/O on one or more of them at any given time. The application will either sit and wait for one of the descriptors to go active, or perhaps dedicate one thread for that purpose while other threads do work.

Such multiplexed and asynchronous I/O is at the basis of the traditional **Posix** system calls `select()` and `poll()`, as well as the **Linux**-only `epoll` system calls which scale the best to large numbers of descriptors.

In order to make `poll()` work on a file descriptor corresponding to a character device, one needs to add the entry point to the `file_operations` table as usual:

```
static struct file_operations mycdrv_fops = {
    .owner = THIS_MODULE,
     ....
    .poll = mycdrv_poll,
};

static unsigned int mycdrv_poll (struct file *file, poll_table * wait);
```

Whenever an application calls `poll()`, `select()` or uses `epoll` this method will be called.

First one must call the function

```
void poll_wait (struct file *filp, wait_head_queue_t *wq, poll_table *wait);
```

for each wait queue whose change of status is to be noted.

Secondly one must return a bit-mask which can be checked to see which (if any) I/O operations are available. A number of flags can be combined in this mask:

Table 19.1: **poll() flags**

Value	Meaning
`POLLIN`	Normal or priority band data can be read without blocking.
`POLLRDNORM`	Normal data can be read. Usually a readable device returns `POLLIN | POLLRDNORM`
`POLLRDBAND`	Priority band data can be read. (This flag is unused.)
`POLLPRI`	High priority out of band data can be read, causing `select()` to report an exception.
`POLLHUP`	Reaching end of file on device.
`POLLERR`	An error has occurred.
`POLLOUT`	The device can be written without blocking.
`POLLWRNORM`	Normal data can be written. Usually a writable device returns `POLLOUT | POLLWRNORM`
`POLLWRBAND`	Priority band data can be written.

An example of an entry point might look like:

```
static unsigned int mycdrv_poll (struct file *file, poll_table * wait)
{
   unsigned int revents = 0;
   poll_wait (file, &wq_read, wait);
   poll_wait (file, &wq_write, wait);

   if ( atomic_read (&data_ready_to_read))
        revents |= POLLIN | POLLRDNORM;
   if ( atomic_read (&data_ready_to_write))
        revents |= POLLOUT | POLLWRNORM;
   return revents;
}
```

19.7 Interrupt Handling in User-Space

Device drivers written in user-space offer certain advantages:

- Potentially better security and stability.

- Keeping the core kernel code base smaller.
- Avoiding some licensing constraints.

Of course not everyone would consider each one of these properties as an advantage.

However, it is already the case that many device drivers are written in user-space either using the `iopl()`, `ioperm()` commands to get application access to I/O ports, or are layered on top of in-kernel lower-level drivers such as those for the parallel, serial or **USB** ports. Such is the case, for example, with drivers for printers and scanners.

What is lacking in terms of infrastructure is a general method of having a user-space driver handle interrupts. The kinds of drivers mentioned above often work in **polling** modes; e.g., the **X** driver checks for mouse activity many times per second by reading I/O ports instead of directly responding to interrupts.

There have been active projects to do this; see **http://lwn.net/Articles/127698/** for a discussion of the effort led by Peter Chubb in which entries are created in the `/proc` file system for each IRQ being dealt with. The user-space driver then sits on that entry either with a `read()` or `poll()` call, until woken up by an interrupt arriving.

There are difficulties such as the possibilities of losing interrupts if multiple interrupts arrive, special problems with sharing interrupts, and trying to avoid too much polling which disturbs true asynchronousness in the interrupt system and can lead to unacceptable latencies.

We will do an exercise in which we implement such a method, using a special device node rather than a /proc entry.

Kernel Version Note

Kernel Version Note

- The **UIO** project (Userspace **I/O**) was accepted in the 2.6.23 kernel
- Rather than working with the `/proc` directory the accepted version creates special device nodes.
- For a full discussion, see **/usr/src/linux/Documentation/DocBook/uio-howto.tmpl.**

19.8 Labs

Lab 1: Using Wait Queues

Generalize the previous character driver to use wait queues,

Have the `read()` function go to sleep until woken by a `write()` function. (You could also try reversing read and write.)

You may want to open up two windows and read in one window and then write in the other window.

Try putting more than one process to sleep, i.e., run your test read program more than once simultaneously before running the write program to awaken them. If you keep track of the **pid**'s you should be able to detect in what order processes are woken.

There are several solutions given:

- Using `wait_event_interruptible()`. You may want to use **atomic** functions for any global variables used in the logical condition.

  There are two solutions with this interface; one that wakes up only one sleeper, one that wakes up all sleepers.
- Using `wait_for_completion()`.
- Using **semaphores**.
- Using **read/write semaphores**.
- Using exclusive waiting on the many readers solution.. How many processes wake up?

If you test with **cat**, **echo**, or **dd**, you may see different results than if you use the supplied simple read/write programs. Why?

Lab 2: Killable Sleep

Modify the `wait_event()` lab to use `wait_event_killable()`. After a reading process goes to sleep, send it a non-fatal signal, such as

```
$ kill -SIGCONT <pid>
```

followed by a kill signal, such as `SIGKILL`.

Lab 3: Using poll()

Take the `wait_event()` solution and extend it to have a `poll()` entry point.

You'll need an application that opens the device node and then calls `poll()` and waits for data to be available.

Lab 4: User-Space Interrupt Handling

Adapt the character driver with polling to handle a shared interrupt.

The read method should sleep until events are available and then deal with potentially multiple events.

The information passed back by the read should include the number of events.

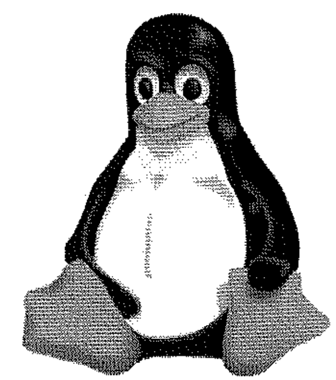
You can reuse the previously written testing program that opens the device node and then sits on it with `poll()` until interrupts arrive.

You can also test it with just using the simple read program, or doing `cat < /dev/mycdrv` and generating some interrupts.

You can probably also implement a solution that does not involve `poll()`, but just a blocking read.

Chapter 20

Interrupt Handling and Deferrable Functions



We'll continue our examination of how the **Linux** kernel handles interrupts, focussing on how the labor is split between top and bottom halves, and what some of the methods are for implementation. We'll investigate the use of deferrable functions, including **tasklets**, **work queues**, and spinning off **kernel threads**. Finally we'll consider the use of threaded interrupt handlers.

20.1 Top and Bottom Halves

Efficient interrupt handlers generally have **top halves** and **bottom halves**.

In the top half, the driver does what must be done as quickly as possible, This may just mean acknowledging the interrupt and getting some data off a device and into a buffer.

In the bottom half, the driver does whatever processing has been deferred. An interrupt handler is not required to have a bottom half.

Top Half

Technically speaking the top half is the interrupt handler. A typical top half:

- Checks to make sure the interrupt was generated by the right hardware; this is necessary for interrupt sharing.
- Clears an **interrupt pending** bit on the interface board.
- Does what needs to be done immediately (usually read or write something to/from the device.) The data is usually written to or read from a device-specific buffer, which has been previously allocated.
- Schedules handling the new information later (in the bottom half.)

Example (using tasklets):

```
static struct my_dat { .... } my_fun_data;

static void t_fun (unsigned long t_arg){ .... }

DECLARE_TASKLET (t_name, t_fun, (unsigned long) &my_data);

static void my_interrupt (int irq, void *dev_id)
{
    top_half_fun ();
    tasklet_schedule (&t_name);
    return IRQ_HANDLED;
}
```

Bottom Half

A bottom half is used to process data while top half is available for dealing with new interrupts. Interrupts are enabled when a bottom half runs. Interrupts can be disabled if necessary, but generally this should be avoided as it goes against the basic purpose of having a bottom half.

The various kinds of bottom halves behave differently:

- **Tasklets** can be run in parallel on different CPUs, although the same tasklet can only be run one at a time. They are never run in process context. Tasklets will run only on the CPU that scheduled them. This leads to better cache coherency, and serialization, as the tasklet can never be run before the handler is done, which leads to better avoidance of race conditions.

- **Work queues** run in process context, and thus sleeping is permitted. Because each work queue has its own thread on each CPU, such sleeping will not block other tasks. A bottom half implemented in this fashion can run on a different CPU than the one that scheduled it.

Depending on the kind of bottom half, they are launched in slightly different ways, but the system always checks whether anything needs to be done after an exception is handled and then runs any queued up bottom halves. This includes:

- After a system call is completed.
- After any other exception is handled.
- After an interrupt is handled.
- When the scheduler selects the kernel process **ksoftirqd** to run.

Another way of implementing a bottom half is through maintaining a `kernel thread`. One starts off the kernel thread upon device initialization or open, and then has it sleep until it has work to do. Scheduling a bottom half then becomes waking up the thread to deal with the work, after which it goes back to sleep. Killing such a thread when a driver is unloaded has to be done with care.

It is not required to have a bottom half; if there is little processing to be done, it may be more efficient to just do it in the top half, rather than incur the overhead of scheduling and launching a bottom half, and having to be careful about synchronization questions.

20.2 Deferrable Functions and softirqs

Deferrable functions perform non-critical tasks at a later (deferred) time, usually as soon as possible. When the functions are run they may be interrupted.

There are two main types: **softirqs** (of which **tasklets** are one kind) which run in interrupt context and are not allowed to go to sleep, and **workqueues**, which run under a pseudo-process context and are allowed to sleep.

There are a number of different kinds of softirqs defined. In order of decreasing priority they are:

Name	Priority	Purpose
`HI_SOFTIRQ`	0	High-priority tasklets.
`TIMER_SOFTIRQ`	1	Scheduled timers.
`NET_TX_SOFTIRQ`	2	Network packet transmission.
`NET_RX_SOFTIRQ`	3	Network packet reception.
`BLOCK_SOFTIRQ`	4	Block device related work.

TASKLET_SOFTIRQ	5	Normal-priority tasklets.
SCHED_SOFTIRQ	6	Used in the **CFS** scheduler.
HRTIMER_SOFTIRQ	7	Used if high resolution timers are present.

The various kinds of deferred functions differ mostly in whether or not they operate in process context and their behaviour on multi-processor systems.

Type	Process Context?	**SMP** Behaviour	**SMP** Serialization
softirq	No	Same ones can be run on different CPUs simultaneously	Run on the CPU that schedules them and must be fully re-entrant.
tasklet	No	Can be run on simultaneously on different CPUs, but not if they use the same `tasklet_struct`.	Run on the CPU that scheduled them and are thus serialized.
workqueue	Yes	Can be run on any CPU.	Can be delayed and serialized.

Softirqs are called, or consumed in either of two ways:

- After an interrupt is serviced the kernel checks if any softirqs are pending; if so it executes them in priority order.
- The kernel thread **ksoftirqd[cpu]** is scheduled in like other processes, and consumes them in like fashion.

The second mechanism is required to prevent priority inversion causing softirq storms when a softirq resubmits itself before finishing. To avoid this, resubmitted deferred functions that exceed a certain backlog are scheduled in like other tasks, so that other work may proceed.

20.3 Tasklets

Tasklets are used to queue up work which can be done at a later time. They are frequently used in interrupt service routines; a typical **top half** does whatever needs to be done to get data off or onto a device and resets it and re-enables interrupts. Further data processing may be done in tasklets while the device is ready for new data.

Tasklets may be run in parallel on multiple CPU systems. However, the same tasklet can not be run at the same time on more than one CPU.

A tasklet is always run on the CPU that scheduled it; among other things this optimizes cache usage. (This however can cause delays which may not be worth the cache savings; **work queues** can be used instead.) As a result, many kinds of race conditions are naturally avoided; the thread that queued up the tasklet must complete before the tasklet actually gets run.

The tasklet code is explained in **/usr/src/linux/include/linux/interrupt.h**. The important data structure is:

```
struct tasklet_struct{
     struct tasklet_struct *next;
     unsigned long state;
     atomic_t count;
     void (*func)(unsigned long);
     unsigned long data;
};
```

The `func` entry is a pointer to the function that will be run, which can have data passed to it through `data`. The `state` entry is used to determine whether or not the tasklet has already been scheduled; if so it can not be done so a second time.

The main macros and functions involving tasklets are:

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

void tasklet_init (struct tasklet_struct *t,
        void (*func)(unsigned long), unsigned long data);

void tasklet_schedule (struct tasklet_struct *t);
void tasklet_enable   (struct tasklet_struct *t);
void tasklet_disable  (struct tasklet_struct *t);
void tasklet_kill     (struct tasklet_struct *t);
```

A tasklet must be initialized before being used, either by allocating space for the structure and calling `tasklet_init()`, or by using the `DECLARE_..()` macros, which take care of both steps although they must be used in the global space.

`DECLARE_TASKLET()` sets up a `struct tasklet_struct name` in an **enabled** state; the second form `DECLARE_TASKLET_DISABLED()` being used means the tasklet can be scheduled but won't be run until the tasklet is specifically enabled.

The `tasklet_kill()` function is used to kill tasklets which reschedule themselves.

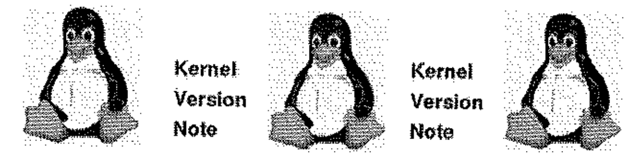
When a tasklet is scheduled, the the inline function `tasklet_schedule()` is called as defined in **/usr/src/linux/include/linux/interrupt.h**:

```
2.6.31: 464 static inline void tasklet_schedule(struct tasklet_struct *t)
2.6.31: 465 {
2.6.31: 466     if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
2.6.31: 467             __tasklet_schedule(t);
2.6.31: 468 }
```

which makes sure the tasklet is not already scheduled, by checking the `state` field of the `tasklet_struct`. Note that failure brings a **quiet** dropping of the tasklet as the function has no return value.

A trivial example:

```
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
#include <linux/slab.h>
#include <linux/init.h>

static void t_fun (unsigned long t_arg);

static struct simp
{
    int i;
    int j;
} t_data;

static DECLARE_TASKLET (t_name, t_fun, (unsigned long)&t_data);

static int __init my_init (void)
{
    printk (KERN_INFO "\nHello: init_module loaded at address 0x%p\n",
            init_module);
    t_data.i = 100;
    t_data.j = 200;
    printk (KERN_INFO " scheduling my tasklet, jiffies= %ld \n", jiffies);
    tasklet_schedule (&t_name);
    return 0;
}

static void __exit my_exit (void)
{
    printk (KERN_INFO "\nHello: cleanup_module loaded at address 0x%p\n",
            cleanup_module);
}

static void t_fun (unsigned long t_arg)
{
    struct simp *datum;
    datum = (struct simp *)t_arg;
    printk (KERN_INFO "Entering t_fun, datum->i = %d, jiffies = %ld\n",
            datum->i, jiffies);
    printk (KERN_INFO "Entering t_fun, datum->j = %d, jiffies = %ld\n",
            datum->j, jiffies);
}

module_init (my_init);
module_exit (my_exit);
```

- There is an ongoing discussion about eliminating tasklets from the **Linux** kernel.
- First, because tasklets run in software interrupt mode, you cannot sleep, refer to user-space, etc., so one has to be quite careful.
- Second, since tasklets run as software interrupts they have higher priority than any other task on the system, and thus can produce uncontrolled latencies in other tasks if they are coded poorly.
- The idea is to replace almost all tasklet uses with workqueues, which run in a sleepable pseudo-user context, and get scheduled like other tasks. A proof of concept implementation in which all tasklets were converted to work queues with a wrapper did not cause terrible problems.
- However, there were developers who were very unhappy with the proposed changes, in particular those who work on network device drivers. In this case testing becomes very laborious.
- If history is any guide the most likely outcome is that the use of tasklets will gradually diminish in that they will be deprecated in new code, and some or a lot of old code will be converted one instance at a time rather than globally. If tasklet use becomes rare it may be eliminated at some point in one fell swoop, but don't lose any sleep waiting for it to happen.

20.4 Work Queues

A **work queue** contains a linked list of tasks which need to be run at a deferred time (usually as soon as possible).

The tasks are run in process context; a kernel thread is run on each CPU in order to launch them. Thus not only is sleeping legal, it will not interfere with tasks running in any other queue. Note that you still can't transfer data to and from user-space as there isn't a real user context to access.

Unlike tasklets, a task run on a work queue may be run on a different processor than the process that scheduled it. Thus they are a good choice when such serialization (and hoping to minimize cache thrashing) is not required, and can lead to faster accomplishment of the deferred tasks.

The code for work queues can be found in **/usr/src/linux/include/linux/workqueue.h** and **/usr/src/linux/kernel/workqueue.c**. The important data structure describing the tasks put on the queue is:

```
typedef void (*work_func_t)(struct work_struct *work);

struct work_struct {
     atomic_long_t data;
     struct list_head entry;
     work_func_t func;
};
```

Here func() points to the function that will be run when the work is done. The other arguments are for internal use and are usually not set directly.

Note that the data entry is used like the state entry for tasklets; if multiple identical work queues are requested, all but the first will be quietly dropped on the floor in the same way.

The earliest implementation of workqueues had an explicit data pointer that was passed to the function. This was modified so that the function now receives a pointer to a work_struct data structure.

In order to pass data to a function, one needs to embed the work_struct in a user-defined data structure and then to pointer arithmetic in order to recover it. An example would be:

```
static struct my_dat
{
    int irq;
    struct work_struct work;
};

static void w_fun (struct work_struct *w_arg)
{
    struct my_dat *data = container_of (w_arg, struct my_dat, work);
    atomic_inc (&bhs[data->irq]);
}
```

A work_struct can be declared and initialized at compile time with:

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

where name is the name of the structure which points to queueing up function() to run. A previously initialized work queue can be initialized and loaded with the two macros:

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

where work has already been declared as a work_struct. The INIT_WORK() macro initializes the list_head linked-list pointer, and PREPARE_WORK() sets the function pointer. The INIT_WORK macro needs to be called at least once, and in turn calls PREPARE_WORK(); it should not be called while a task is already in the work queue.

While it is possible to set up your own work queue for just your own tasks, in most cases a default work queue (named **events**) will suffice, and is easier to use. Tasks are added to and flushed from this queue with the functions:


```
int schedule_work (struct work_struct *work);
void flush_scheduled_work (void);
```

flush_scheduled_work() is used when one needs to wait until all entries in a work queue have run,

Note that these are the only work queue functions that are exported to all modules; the others are exported only to **GPL**-compliant modules. Thus creating your own work queue and using it is reserved only for **GPL**-licensed code.

A work queue can be created and destroyed with:

```
struct workqueue_struct *create_workqueue (const char *name);
void destroy_workqueue (struct workqueue_struct *wq);
```

where name is up to 10 characters long and is the command listed for the thread, and the struct workqueue_struct describes the work queue itself (which one never needs to look inside), Note that destroy_workqueue() flushes the queue before it returns.

Adding a task to the work queue, and flushing it is done with:

```
int queue_work (struct workqueue_struct *wq, struct work_struct *work);
void flush_workqueue (struct workqueue_struct *wq);
```

It is possible to postpone workqueue execution for a specified timer interval using:

```
struct delayed_work { struct work_struct_work, struct timer_list timer;}
int schedule_delayed_work (struct delayed_work *work, unsigned long delay);
int cancel_delayed_work (struct delayed_work *work);

DECLARE_WORK(name, void (*function)(void *));
INIT_WORK(   struct delayed_work *work, void (*function)(void *));
PREPARE_WORK(struct delayed_work *work, void (*function)(void *));
```

where delay is expressed in jiffies. One can use cancel_delayed_work() to kill off a pending delayed request.

One has to be careful when taking advantage of a task's ability to sleep on a workqueue; when it sleeps, no other pending task on the queue can run until it wakes up!

The workqueue implementation also provides a method of ensuring a function runs in process context:

```
typedef void (*work_func_t)(struct work_struct *work);
struct execute_work {
        struct work_struct work;
};

int execute_in_process_context (work_func_t fn, struct execute_work *ew);
```

If this function is called from process context it will return a value of 0 and fn(data) will be run immediately. If this function is called from interrupt context it will return a value of 1 and the function will be called with

```
schedule_work(&ew_work);
```

20.5 Creating Kernel Threads

kernel threads of execution differ in many important ways from those that operate on behalf of a process. For one thing they always operate in kernel mode.

The functions and macros for creating and stopping kernel threads are given in **/usr/src/linux/include/linux/kthread.h**:

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

struct task_struct *kthread_run (int (*threadfn)(void *data) void *data,
                                 const char namefmt[], ...);
struct task_struct *kthread_create (int (*threadfn)(void *data) void *data,
                                 const char namefmt[], ...);
void kthread_bind (struct task_struct *k, unsigned int cpu);
int kthread_stop (struct task_struct *k);
int kthread_should_stop (void);
```

The created thread will run `threadfn(data)`, which will use `namefmt` and any succeeding arguments to create its name as it will appear with the **ps** command.

The function `kthread_create()` initializes the process in a sleeping state; usually one will want to use the `kthread_run()` macro which follows this with a call to `wake_up_process()`. However, one may want to call `kthread_bind()` first, which will bind the thread to a particular cpu.

Terminating the thread is done with `kthread_stop()`. This sets `kthread_should_stop()`, wakes the thread and waits for it to exit. For example one might execute a loop such as:

```
do {  .... } while (!kthread_should_stop());
```

where the loop will probably include sleeping, and then issue a call to `kthread_stop()` from an exit routine.

Kernel threads can only be created from process context as their implementation can block while waiting for resources. Calling from atomic context will lead to a kernel crash.

- An older function

  ```
  int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags);
  ```

  is still used in many places in the kernel. It is more complicated to use and requires more work to accomplish successful termination. It should not be used in new code.

20.6 Threaded Interrupt Handlers

The 2.6.30 kernel introduced a new method of writing interrupt handlers in which the bottom half is taken care by a scheduled thread. This feature arose in the **realtime** kernel tree and unsurprisingly has as its goal reducing latencies and the amount of time interrupts may need to be disabled.

The API is only slightly different than that used in the normal interrupt handler; an IRQ is now requested with the function:

```
int request_threaded_irq (unsigned int irq, irq_handler_t handler,
         irq_handler_t thread_fn, unsigned long flags, const char *name, void *dev);
```

the new aspect being the third argument, `thread_fn` which is essentially a bottom half. There is a also a new return value for the top half, `IRQ_WAKE_THREAD`, that should be used when the threaded bottom half is being used.

Thus the top half is called in a hard interrupt context, and and must first check whether the interrupt originated in its device. If not it returns `IRQ_NONE`; otherwise it returns `IRQ_HANDLED` if no further processing is required, or `IRQ_WAKE_THREAD` if the thread function needs to be invoked. In this case it should have disabled the interrupt on the device level.

While this method has not yet percolated into interrupt handlers, eventually it might replace tasklets and work queues in most arenas. One can expect to see a gradual adoption of this method, especially in new drivers.

20.7 Labs

Lab 1: Deferred Functions

Write a driver that schedules a deferred function whenever a `write()` to the device takes place.

Pass some data to the driver and have it print out.

Have it print out the `current->pid` field when the tasklet is scheduled, and then again when the queued function is executed.

Implement this using:

- **tasklets**
- **work queues**

You can use the same testing programs you used in the sleep exercises.

Try scheduling multiple deferred functions and see if they come out in LIFO or FIFO order. What happens if you try to schedule the deferred function more than once?

Lab 2: Shared Interrupts and Bottom Halves

Write a module that shares its IRQ with your network card. You can generate some network interrupts either by browsing or pinging.

Make it use a top half and a bottom half.

Check `/proc/interrupts` while it is loaded.

Have the module keep track of the number of times the interrupt's halves are called.

Implement the bottom half using:

- **tasklets**.
- **work queues**
- **A background thread** which you launch during the module's initialization, which gets woken up anytime data is available. Make sure you kill the thread when you unload the module, or it may stay in a zombie state forever.

For any method you use does, are the bottom and top halves called an equal number of times? If not why, and what can you do about it?

Lab 3: Producer/Consumer

You may have noticed that you lost some bottom halves. This will happen when more than one interrupt arrives before bottom halves are accomplished. For instance, the same tasklet can only be queued up twice.

Write a bottom half that can "catch up"b; i.e., consume more than one event when it is called, cleaning up the pending queue. Do this for at least one of the previous solutions.

Lab 4: Sharing All Interrupts, Bottom Halves

Extend the solution to share all possible interrupts, and evaluate the consumer/producer problem.

Lab 5: Sharing All Interrupts, Bottom Halves, Producer/Consumer Problem

Find solutions for the producer/consumer problem for the previous lab.

Lab 6: Threaded Interrupt Handlers

If you are running a kernel version 2.6.30 or later, solve the producer/consumer problem with a threaded interrupt handler.

There are two types of solutions presented, one for just one shared interrupt, one sharing them all, with the same delay parameter as used in the earlier exercises.

Lab 7: Executing in Process Context

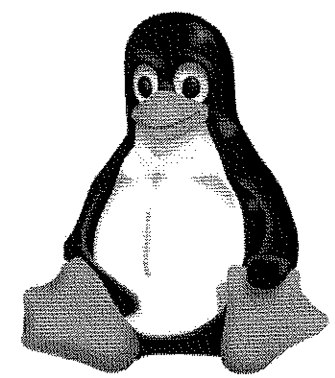
Write a brief module that uses `execute_in_process_context()`. It should do this first in process context (during initialization would be sufficient) and then in an interrupt routine.

You can adapt the simplest shared interrupt lab module to do this.

Make sure you print out the return value in order to see whether it just ran the function directly, or from a work queue.

Chapter 21

Hardware I/O



We'll see how **Linux** communicates with data buses and I/O Ports, uses memory barrier, how device drivers register and unregister them, read and write to them, and slow them down. We'll see how to read and write to memory mapped devices. We'll also briefly consider how to access I/O Ports from user-space.

21.1 Buses and Ports

Computers require data paths for the flow of information between the processor, memory, and the various I/O devices and other peripherals. These data paths are known as the **bus**, of which there are several kinds:

- A **data bus** is a group of lines that do parallel data transfer; on the Pentium data buses are 64-bit wide.
- An **address bus** transmits addresses; on the Pentium address buses are 32-bit wide.
- A **control bus** transmits control information, such as whether the bus can allow data to go between a CPU and RAM, or between a CPU and an I/O device, or whether a read or write is to be performed.

A bus connecting a CPU to an I/O device is called an **I/O bus**; **x86** CPUs use 16 of 32 address lines to address I/O devices, and 8, 16, or 32 out of the 64 data lines to transfer data. The I/O bus is connected to each I/O device through a combination of **I/O ports**, **interfaces**, and **device controllers**.

The buses can be of various types such as **ISA, EISA, PCI** and **MCA**. We'll restrict our attention to **ISA** and **PCI**.

Controlling peripheral devices generally involves reading and writing to **registers** on the device. When we talk about **I/O ports**, we are referring to the consecutive addresses, or registers.

Exactly how these ports are accessed depends on the CPU. All are attached to some kind of peripheral bus, but some CPUs, such as the **x86** actually have distinct read and write lines and special CPU instructions to access these memory locations. On other architectures, memory is memory. Portable code uses the same basic functions regardless of the architecture, although the implementation of the functions may differ.

On the **x86** architecture this I/O address space is 64K in length; ports can be addressed as individual 8-bit ports, while any two consecutive 8-bit ports can be treated as a 16-bit port, and four consecutive 8-bit ports can be treated as a 32-bit port. Thus, to be more precise, you can have 64K 8-bit ports, or 32K 16-bit ports, 16K 32-bit ports, or some other combination. The 16-bit and 32-bit ports should be aligned on 16-bit and 32-bit boundaries.

21.2 Memory Barriers

Operations on I/O registers differ in some important ways from normal memory access. In particular, there may be so-called **side-effects**. These are generally due to compiler and hardware optimizations.

These optimizations can cause reordering of instructions. In conventional memory reads and writes there is no problem; a write always stores a value and a read always returns the last value written.

However, for I/O ports problems can result because the CPU cannot tell when a process depends on the order of memory access. In other words, because of reading or writing an I/O register, devices may initiate or respond to various actions.

Therefore, a driver must make sure no caching is performed and no reordering occurs. Otherwise problems which are difficult to diagnose, and are rare or intermittent, may result.

The solution is to use appropriate **memory barrier** functions when necessary. The necessary functions are defined in and indirectly included from **/usr/src/linux/arch/x86/include/asm/system.h** and are:

```
void barrier (void)

void rmb (void)
void wmb (void)
void mb (void)

void smp_rmb (void)
void smp_wmb (void)
void smp_mb (void)
```

The `barrier()` macro causes the compiler to store in memory all values currently modified in a CPU register, to read them again later when they are needed. This function does not have any effect on the hardware itself.

The other macros put hardware memory barriers in the code; how they are implemented depends on the platform. `rmb()` forces any reads before the barrier to complete before any reads done after the barrier; `wmb()` does the same thing for writes, while `mb()` does it for both reads and writes.

The versions with `smp_` insert hardware barriers only on multi-processor systems; on single CPU systems they expand to a simple call to `barrier()`.

A simple example of a use of a write barrier would be:

```
io32write (direction, dev->base + OFF_DIR);
io32write (size, dev->base + OFF_SIZE);
wmb();
io32write (value, dev->base + OFF_GO);
```

Most architectures define convenience macros, which combine setting a value with invoking a memory barrier. In the simplest form they look like:

```
#define set_mb(var, value)  do { var = value; mb();  } while (0)
#define set_wmb(var, value) do { var = value; wmb(); } while (0)
#define set_rmb(var, value) do { var = value; rmb(); } while (0)
```

Memory barriers may cause a performance hit and should be used with care. One should only use the specific form needed. For instance on **x86** the write memory barrier does nothing as writes are not reordered. However, reads may be reordered, so you should not use `mb()` if `wmb()` would suffice.

21.3 Registering I/O Ports

Before we can access the I/O ports, the kernel has to **register** their use, although there is nothing at the hardware level to enforce this which can lead to many bugs and system crashes.

Linux uses the following functions, defined in `/usr/src/linux/kernel/resource.c` for requesting and releasing I/O ports:

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

struct resource *request_region (unsigned long from, unsigned long extent, const char *name);
void release_region (unsigned long from, unsigned long extent);
```

In these functions the argument `from` is the base address of the I/O region, the argument `extent` is the number of ports, or addresses, and the argument `name` is the name that will appear in **/proc/ioports** as having claimed the region.

These functions are usually called when initializing or unloading a device. `request_region()` will reserve (register) the region, while `release_region()` will free (unregister) it. If any part of the range of I/O ports has already been reserved, the request will fail.

Note: there is no ***enforcement*** here; i.e., if you try to access a given I/O port without checking or requesting it, nothing will stop you.

Example:

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

static int my_dev_detect( unsigned long port_addr,
                          unsigned long extent )
{
   if( !request_region( port_addr, extent, "my_dev")
       return -EBUSY ;       /* the port is busy */

   if( mydrv_probe(port_addr,extent) != 0 ) {
        release_region(port_addr, extent);
        return -ENODEV    /* can't find the device */
   }
   return 0 ;
}
```

21.4 Resource Management

You may have noticed that the `request_region()` function returns a pointer to a structure of type:

```
struct resource {
   const char *name;
   unsigned long start, end;
   unsigned long flags;
   struct resource *parent, *sibling, *child;
};
```

which represents a layer of abstraction: a **resource** is a portion of some entity that can be exclusively assigned to a device driver. In this case the resource is the range of I/O ports.

In this structure, the `name` element describes the resource's **owner**, the `start` and `end` elements give the range of the resource (their precise meanings depending on what the resource is), the `flags`

element can be used to describe various attributes, and the `parent`, `sibling` and `child` fields place the structure in a **resource tree** which contains all resources of the same kind.

Thus all resources referring to I/O Ports are in the tree stemming from the `ioport_resource` head node. Management of the I/O ports can be done through the functions:

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

int request_resource (struct resource *root, struct resource *new);
int release_resource (struct resource *new);
```

instead of through the `*_region()` functions described previously, which are just wrappers for the `*_resource()` functions.

Example:

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

static struct resource my_resource = { "my_dev",};

static int my_dev_detect( unsigned long port_addr,
                          unsigned long extent )
{
   unsigned long end = port_addr + extent;
   my_resource.start = port_addr;
   my_resource.end   = end;

   if( !request_resource (&ioport_resource, &my_resource))
       return -EBUSY;      /* the port is busy */

   if( mydrv_probe(port_addr,extent) != 0 ) {
       release_resource(&my_resource);
       return -ENODEV    /* can't find the device */
   }
   return 0 ;
}
```

- For specific buses there are optional convenience functions that take care of allocating I/O resources and remapping them. For instance for **PCI** one can use the following functions (and others) defined in **/usr/src/linux/drivers/pci/pci.c**:

```
int pci_request_region  (struct pci_dev *pdev, int bar, const char *res_name);
int pci_request_regions (struct pci_dev *pdev, const char *res_name);
int pci_release_regions (struct pci_dev *pdev);
void pci_release_region (struct pci_dev *pdev, int bar);
```

- You can request either a particular **bar** (Base Address Register) associated with the device, where bar can range from 0 to 5, or all regions associated with a device.
- The res_name argument is what shows up under **/proc/iomem**.

21.5 Reading and Writing Data from I/O Registers

The following macros are defined in asm/io.h, and give the ability to read and write 8-bit, 16-bit, and 32-bit ports, once or multiple times:

Reading:

```
unsigned char  inb  (unsigned long port_address);
unsigned short inw  (unsigned long port_address);
unsigned long  inl  (unsigned long port_address);
void           insb (unsigned long port_address, void *addr, unsigned long count);
void           insw (unsigned long port_address, void *addr, unsigned long count);
void           insl (unsigned long port_address, void *addr, unsigned long count);
```

Writing:

```
void outb  (unsigned char b,  unsigned long port_address);
void outw  (unsigned short w, unsigned long port_address);
void outl  (unsigned long l,  unsigned long port_address);
void outsb (unsigned port_address, void *addr, unsigned long count);
void outsw (unsigned port_address, void *addr, unsigned long count);
void outsl (unsigned port_address, void *addr, unsigned long count);
```

Note that the long functions give only 32-bit operations; there is no 64-bit data path even on 64-bit platforms.

The functions above that take the **count** argument do not write to a range of addresses; they write only to the one port address, but they loop efficiently around the operation.

All these functions do I/O in **little-endian** order, and do any necessary byte-swapping.

Reading and writing I/O ports may require the use of **memory barriers**, which we previously discussed.

Example:

```
outb(MSE_READ_X_LOW, MSE_CONTROL_PORT);

dx = (inb(MSE_DATA_PORT) & 0xf);
```

21.6 Slowing I/O Calls to the Hardware

Pausing functions can be used to handle I/O to slow devices. They have the same form as the usual read/write functions, but with the _p appended to their names; i.e., inb_p(), outw_p(), etc., and are defined in **/usr/src/linux/arch/x86/include/asm/io.h** through some very complicated macro magic.

These functions insert a small delay after the I/O instruction if another such function follows. They should not be necessary except for very old **ISA** hardware.

While there is no precise documentation on the length of the introduced delay, a heuristic test can be applied with the following calibration program:

```
/* IOPORT FROM 0x200 to 0x240 is free on my system (64 bytes) */
#define IOSTART  0x200
#define IOEXTEND 0x40

#include <linux/module.h>
#include <linux/ioport.h>
#include <linux/jiffies.h>
#include <linux/io.h>
#include <linux/init.h>

#define NLOOP 1000000          /* should be a multiple of millions */
#define BILL 1000000000        /* make the time in nanoseconds) */

static int __init my_init (void)
{
    int j;
    unsigned long ultest = (unsigned long)1000;
    unsigned long jifa, jifb, jifc, jifd;

    if (!request_region (IOSTART, IOEXTEND, "my_ioport")) {
        printk (KERN_INFO "the IO REGION is busy, quitting\n");
        return -EBUSY;
    }
    printk (KERN_INFO " requesting the IO region from 0x%x to 0x%x\n",
            IOSTART, IOSTART + IOEXTEND);

    /* get output delays */

    jifa = jiffies;
```

```
    for (j = 0; j < NLOOP; j++)
        outl (ultest, IOSTART);
    jifb = jiffies;
    jifc = jiffies;
    for (j = 0; j < NLOOP; j++)
        outl_p (ultest, IOSTART);
    jifd = jiffies;
    printk (KERN_INFO
            "outl: nsec/op=%ld  outl_p: nsec/op=%ld   nsec delay/op=%ld\n",
            (jifb - jifa) * (BILL / NLOOP) / HZ,
            (jifd - jifc) * (BILL / NLOOP) / HZ,
            ((jifd - jifc) - (jifb - jifa)) * (BILL / NLOOP) / HZ);

    /* get input delays */

    jifa = jiffies;
    for (j = 0; j < NLOOP; j++)
        ultest = inl (IOSTART);
    jifb = jiffies;
    jifc = jiffies;
    for (j = 0; j < NLOOP; j++)
        ultest = inl_p (IOSTART);
    jifd = jiffies;
    printk (KERN_INFO
            " inl: nsec/op=%ld   inl_p: nsec/op=%ld   nsec delay/op=%ld\n",
            (jifb - jifa) * (BILL / NLOOP) / HZ,
            (jifd - jifc) * (BILL / NLOOP) / HZ,
            ((jifd - jifc) - (jifb - jifa)) * (BILL / NLOOP) / HZ);
    return 0;
}
static void __exit my_exit (void)
{
    printk (KERN_INFO " releasing  the IO region from 0x%x to 0x%x\n",
            IOSTART, IOSTART + IOEXTEND);
    release_region (IOSTART, IOEXTEND);
}

module_init (my_init);
module_exit (my_exit);
MODULE_LICENSE ("GPL v2");
```

Running this on a variety of different CPU's gives extra delay of about a microsecond per operation.

21.7 Allocating and Mapping I/O Memory

Non-trivial peripheral devices are almost always accessed though on-board memory which is remapped and made available to the processor over the bus. These memory locations can be used as buffers, or behave as I/O ports which have side effects associated with I/O operations.

Exactly how these memory regions are accessed is quite architecture-dependent. However, **Linux** hides the platform dependence by using a universal interface. While some architectures permit direct dereferencing of pointers for these regions, one should never attempt this.

There are three essential steps in using these regions: allocation, remapping, and use of the appropriate read/write functions.

Before such a memory region can be used it must be allocated (and eventually freed) with:

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

which work on a region of `len` bytes, extending from address `start`, and using `name` to describe the entry created in **/proc/iomem**. The starting address is a characteristic of the device; e.g., for **PCI** devices it may be read from a configuration register, or obtained from the function `pci_resource_start()`.

One can not directly use the pointer to the `start` address; instead one must remap and eventually unmap it with:

```
#include <linux/io.h>
void *ioremap (unsigned long phys_addr, unsigned long size);
void iounmap (void *addr);
```

Furthermore, one should refer to this memory only with the functions to be described next, not direct pointer dereferencing.

Occasionally, one may find it convenient to use the following functions to associate I/O registers, or ports, with I/O memory:

```
#include <asm-generic/iomap.h>

void *ioport_map (unsigned long port, unsigned int count);
void ioport_unmap (void *addr);
```

By using these functions I/O ports appear as memory. These ports will have to be reserved as usual before this is done. After doing this, access is obtained with the read/write functions to be discussed next.

Once again there are bus-specific optional convenience functions, such as

```
void *pci_iomap (struct pci_dev *dev, int bar, unsigned long maxlen);
void pci_iounmap (struct pci_dev *dev, void __iomem * addr);
```

defined in **/usr/src/linux/lib/iomap.c**.

Note these functions do not request the memory regions; that must be done separately.

21.8 Accessing I/O Memory

Reading and writing from remapped I/O memory is done with the following functions:

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

```
unsigned int ioread8  (void *addr);
unsigned int ioread16 (void *addr);
unsigned int ioread32 (void *addr);

void iowrite8  (u8  val, void *addr);
void iowrite16 (u16 val, void *addr);
void iowrite32 (u32 val, void *addr);
```

The addr argument should point to an address obtained with `ioremap()` (with perhaps an offset), with the read functions returning the value read.

Reading and writing multiple times can be done with

```
void ioread8_rep  (void *addr, void *buf, unsigned long count);
void ioread16_rep (void *addr, void *buf, unsigned long count);
void ioread32_rep (void *addr, void *buf, unsigned long count);

void iowrite8_rep  (void *addr, void *buf, unsigned long count);
void iowrite16_rep (void *addr, void *buf, unsigned long count);
void iowrite32_rep (void *addr, void *buf, unsigned long count);
```

These functions do repeated I/O on addr, not to a range of addresses, reading from or writing to the kernel address pointed to by buf.

Most 64-bit architectures also have 64-bit reads and writes, with the functions:

```
u64 readq (address);
void writeq (u64 val, address);
```

used in an obvious way, where the **q** stands for **quad**. Note there are no `ioread64()`, `iowrite64()` functions at this time.

Working directly with a block of memory can be done with

```
void memset_io (void *addr, u8 val, unsigned int count);
void memcpy_io   (void *dest, void *source, unsigned int count);
void memcpy_toio (void *dest, void *source, unsigned int count);
```

The above functions do I/O in **little-endian** order, and do any necessary byte-swapping., except for the `mem...()` ones which simply work with byte streams and do no swapping.

The older I/O functions:

```
unsigned char  readb (address);
unsigned short readw (address);
unsigned long  readl (address);

void writeb (unsigned char val,  address);
void writew (unsigned short val, address);
void writel (unsigned long val,  address);
```

are deprecated, although they will still work. They are not as safe as the newer functions as they do not do as thorough type checking.

21.9 Access by User - ioperm(), iopl(), /dev/port

I/O Ports can also be accessed from user-space. This is a technique often used by *user-space drivers*, such as the various **X**-servers. Applications doing this must be run as **root**. Thus they are dangerous to use for both stability and security.

One method is to use the functions:

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

int ioperm (unsigned long from, unsigned long num, int turn_on);
int iopl (int level);
```

`ioperm()` gets permission for individual ports, for `num` bytes from the port address `from`, enabling if `turn_on = 1`.

Only the first `0x3ff` ports can be accessed this way; for larger values you have to use `iopl()`, which gets permission for the entire I/O space.

The **level** argument can range from 0 to 3. Ring levels less than or equal to this value will be given access to the I/O Ports; thus a value `level=3` lets normal user applications (in Ring 3) have access to I/O Ports.

When using these facilities you can use `inb()`, `outb()` etc., functions from user-space. This requires compilation with optimization turned on to ensure expansion of inline functions.

Another method is to use the `/dev/port` device node. One merely seeks to the correct offset and uses normal read and write functions. This back door is considered quite dangerous but has often been used in legacy applications.

21.10 Labs

Lab 1: Accessing I/O Ports From User-Space

Look at **/proc/ioports** to find a free I/O port region. One possibility to use the first parallel port, usually at 0x378, where you should be able to write a 0 to the register at the base address, and read the next port for status information.

Try reading and writing to these ports by using two methods:

- `ioperm()`
- `/dev/port`

Lab 2: Accessing I/O Ports

Look at **/proc/ioports** to find a free I/O port region.

Write a simple module that checks if the region is available, and requests it.

Check and see if the region is properly registered in **/proc/ioports**.

Make sure you release the region when done.

The module should send some data to the region, and read some data from it. Do the values agree? If not, why?

Note: there are two solutions given, one for the older *region* API, one for the newer *resource* API.

Lab 3: Remapping I/O Ports

Alter your solution to use `ioport_map()` and the proper reading and writing functions.

Lab 4: Serial Mouse Driver

Attach a generic serial mouse using the Microsoft protocol to a free serial port.

Depending on which serial port you have chosen, you'll have to know the relevant IRQ and base register address; i.e.,

Table 21.2: **Serial mouse nodes and registers**

Port	Node	IRQ	IOPORT
com1	/dev/ttyS0	4	0x03f8-0x03ff
com2	/dev/ttyS1	3	0x02f8-0x02ff
com3	/dev/ttyS2	4	0x03e8-0x03ef
com4	/dev/ttyS3	3	0x02e8-0x02ef

You will need to view the man page for **mouse**, which says in part:

```
Microsoft protocol

    The Microsoft  protocol uses 1 start bit, 7 data bits, no
    parity and one stop bit at the  speed  of  1200  bits/sec.
    Data  is  sent  to  RxD  in 3-byte packets.  The dx and dy
    movements are sent as two's-complement, lb  (rb)  are  set
    when the left (right) button is pressed:

         byte  d6   d5   d4   d3   d2   d1   d0
            1   1   lb   rb   dy7  dy6  dx7  dx6
            2   0   dx5  dx4  dx3  dx2  dx1  dx0
            3   0   dy5  dy4  dy3  dy2  dy1  dy0
```


You will also have to take a good look at **/usr/src/linux/include/linux/serial_reg.h** which gives the various UART port assignments (as offsets from the base register) and the symbolic definitions for the various control registers.

Your driver should contain:

- An **interrupt** routine which prints out the consecutive number of the interrupt (i.e., keep a counter), the `dx` and `dy` received, and the cumulative `x` and `y` positions.
- A **read** entry that reports back to user-space the current `x` and `y` positions of the mouse.
- An **ioctl** entry that can zero out the cumulative `x` and `y` positions.

You'll have to write a user-space application to interact with your driver, of course.

The trickiest part here is initialization of the mouse. You will have to initialize the outgoing registers properly to enable interrupts, the FIFO register, the Line Control Register, and the Modem Control Register.

The worst part of doing this is to set the baud rate. You can do this directly in your driver but it is not easy to figure out. A work around is to run the command (as a script perhaps):

```
gpm -M -D -t ms -m /dev/ttyS0 -V
```

and then kill it, which should set things up ok. (On some PC's this step is unnecessary, either due to **BIOS** differences, or to the way **Linux** has been booted.) It is also possible to do this in other ways, such as using the system command `setserial` or, depending on how you handle the next step, merely opening **/dev/ttyS?** from a user-space application. You can also try

```
stty -F /dev/ttyS0 ospeed 1200 ispeed 1200
```

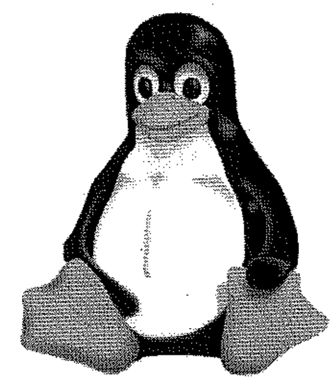
If you get hung up on setting the speed, or decoding the bytes, the solutions contain ***hint*** files that contain the code for doing these steps.

While you can do this exercise under **X**, it will probably cause fewer headaches to do it at a console, as **X** has some ideas about how to handle the mouse.

EXTRA: Construct a fully functional serial mouse driver, and use it under X. Note to do this you'll have to modify `/etc/X11/xorg.conf` to point to your driver and the protocol. The read entry should deliver the latest raw 3 byte packet, and pad with zeros for any more than 3 bytes requested. You'll have to be careful with things like making sure the packet is not reset while you are reading, etc.

Chapter 22

PCI



We'll see how **Linux** uses **PCI** devices, and describe the various functions used to find and manipulate them. We'll also consider the newer **PCI Express** standard.

22.1 What is PCI?

PCI stands for Peripheral Component Interconnect. It replaces **ISA** (Industry Standard Architecture) with three main goals:

- Better performance transferring data between CPU and peripherals.
- Platform-independent as possible.

- Simplify adding and removing peripherals.

Information on the **PCI** devices currently installed on the system can be obtained with the command:

```
$ lspci -v

00:00.0 Host bridge: Intel Corporation 4 Series Chipset DRAM Controller
                     (rev 02)
        Subsystem: ASUSTeK Computer Inc. Unknown device 82d3
        Flags: bus master, fast devsel, latency 0
        Capabilities: [e0] Vendor Specific Information
....
00:1a.7 USB Controller: Intel Corporation 82801JI (ICH10 Family) USB2 EHCI
                        Controller #2 (prog-if 20 [EHCI])
        Subsystem: ASUSTeK Computer Inc. Unknown device 82d4
        Flags: bus master, medium devsel, latency 0, IRQ 18
        Memory at f9fffc00 (32-bit, non-prefetchable) [size=1K]
        Capabilities: [50] Power Management version 2
        Capabilities: [58] Debug port
        Capabilities: [98] Vendor Specific Information
....
02:00.0 Ethernet controller: Marvell Technology Group Ltd. 88E8056 PCI-E
                             Gigabit Ethernet Controller (rev 12)
        Subsystem: ASUSTeK Computer Inc. Unknown device 81f8
        Flags: bus master, fast devsel, latency 0, IRQ 29
        Memory at fe9fc000 (64-bit, non-prefetchable) [size=16K]
        I/O ports at c800 [size=256]
        Expansion ROM at fe9c0000 [disabled] [size=128K]
        Capabilities: [48] Power Management version 3
        Capabilities: [50] Vital Product Data
        Capabilities: [5c] Message Signalled Interrupts: 64bit+ Queue=0/0
                      Enable+
        Capabilities: [e0] Express Legacy Endpoint IRQ 0
        Capabilities: [100] Advanced Error Reporting
...
01:00.0 VGA compatible controller: nVidia Corporation GeForce 8400 GS
                                   (rev a1) (prog-if 00 [VGA controller])
        Subsystem: ASUSTeK Computer Inc. Unknown device 8278
        Flags: bus master, fast devsel, latency 0, IRQ 16
        Memory at fd000000 (32-bit, non-prefetchable) [size=16M]
        Memory at d0000000 (64-bit, prefetchable) [size=256M]
        Memory at fa000000 (64-bit, non-prefetchable) [size=32M]
        I/O ports at bc00 [size=128]
        [virtual] Expansion ROM at fe8e0000 [disabled] [size=128K]
        Capabilities: [60] Power Management version 3
        Capabilities: [68] Message Signalled Interrupts: 64bit+
                      Queue=0/0 Enable-
        Capabilities: [78] Express Endpoint IRQ 0
        Capabilities: [100] Virtual Channel
        Capabilities: [128] Power Budgeting
        Capabilities: [600] Unknown (11)
```


The information returned about each device comes from its **configuration register**, a 256-byte address space on the board, which is read during boot and **PCI** bus initialization. Note the first three fields which identify a **PCI** device:

Table 22.1: **PCI features**

bus number:	256 are permitted, but most PC's have only a few.
device number:	Each bus can have up to 32 devices.
function number:	Each device can have up to 8 functions.

The **PCI** Chapter of ***Corbet, Rubini and Kroah-Hartman*** book gives more detailed information about the layout of the configuration register and the fields incorporated in it.

Under **Linux**, detection of **PCI** devices is done at boot; the configuration registers are located and read and their contents are placed in memory in a linked list of data structures. (We've left hot-swappable devices out of this discussion.)

When a **PCI** system boots devices initially have no memory, I/O ports, IRQ's, etc. assigned. The System **BIOS** finds safe assignments for these resources before a device driver can gain access to the resource; they will then be obtainable from the configuration register. Any firmware on the device is read after the **BIOS** scan.

A view of the devices on the bus can easily be obtained by looking at **sysfs**. For example, picking the sixth bus, third device slot, first function:

```
$ ls -lF  /sys/bus/pci/devices/0000:05:02.0
total 0
-rw-r--r-- 1 root root   4096 May 27 07:15 broken_parity_status
lrwxrwxrwx 1 root root      0 May 27 02:14 bus -> ../../../../bus/pci/
-r--r--r-- 1 root root   4096 May 27 07:15 class
-rw-r--r-- 1 root root    256 May 27 07:15 config
-r--r--r-- 1 root root   4096 May 27 07:15 device
lrwxrwxrwx 1 root root      0 May 27 02:14 driver -> \
                                      ../../../../bus/pci/drivers/skge/
-rw------- 1 root root   4096 May 27 07:15 enable
-r--r--r-- 1 root root   4096 May 27 07:15 irq
-r--r--r-- 1 root root   4096 May 27 07:15 local_cpulist
-r--r--r-- 1 root root   4096 May 27 07:15 local_cpus
-r--r--r-- 1 root root   4096 May 27 07:15 modalias
-rw-r--r-- 1 root root   4096 May 27 07:15 msi_bus
lrwxrwxrwx 1 root root      0 May 27 07:15 net:eth0 ->
                                          ../../../../class/net/eth0/
drwxr-xr-x 2 root root      0 May 27 07:15 power/
-r--r--r-- 1 root root   4096 May 27 07:15 resource
-rw------- 1 root root  16384 May 27 07:15 resource0
-rw------- 1 root root    256 May 27 07:15 resource1
-r-------- 1 root root 131072 May 27 07:15 rom
```

```
lrwxrwxrwx 1 root root      0 May 27 07:15 subsystem ->
                                            ../../../../bus/pci/
-r--r--r-- 1 root root   4096 May 27 07:15 subsystem_device
-r--r--r-- 1 root root   4096 May 27 07:15 subsystem_vendor
-rw-r--r-- 1 root root   4096 May 27 02:14 uevent
-r--r--r-- 1 root root   4096 May 27 07:15 vendor
-rw------- 1 root root  32768 May 27 07:15 vpd
```

The driver may need to read and/or write to three address spaces: memory, port, and configuration, and we'll discuss each.

22.2 PCI Device Drivers

One registers and unregisters a **PCI** device driver with:

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

int  pci_register_driver   (struct pci_driver *);
void pci_unregister_driver (struct pci_driver *);
```

The registration/de-registration functions are normally called in your initialization and cleanup functions.

The function `pci_register_driver()` returns the number of **PCI** devices claimed by the driver when registering; even if this is 0, the driver will need to be unregistered. If PCI is not configured, this function will return 0.

The registration functions use a data structure of type `pci_driver`:

```
2.6.31: 473 struct pci_driver {
2.6.31: 474     struct list_head node;
2.6.31: 475     char *name;
2.6.31: 476     const struct pci_device_id *id_table;   /* must be non-NULL for probe to be
                                                           called */
2.6.31: 477     int  (*probe)  (struct pci_dev *dev, const struct pci_device_id *id);   /*
                                                                  New device inserted */
2.6.31: 478     void (*remove) (struct pci_dev *dev);   /* Device removed (NULL if not a
                                                           hot-plug capable driver) */
2.6.31: 479     int  (*suspend) (struct pci_dev *dev, pm_message_t state);      /* Device
                                                                        suspended */
2.6.31: 480     int  (*suspend_late) (struct pci_dev *dev, pm_message_t state);
2.6.31: 481     int  (*resume_early) (struct pci_dev *dev);
2.6.31: 482     int  (*resume) (struct pci_dev *dev);                  /* Device woken up */
2.6.31: 483     void (*shutdown) (struct pci_dev *dev);
2.6.31: 484     struct pci_error_handlers *err_handler;
2.6.31: 485     struct device_driver    driver;
2.6.31: 486     struct pci_dynids dynids;
2.6.31: 487 };
```

Important elements of the data structure are:

Table 22.2: **pci_driver structure elements**

Field	Meaning
`id_table`	Points to a table of device ID's of interest to the driver. Usually this will be exported with the macro `MODULE_DEVICE_TABLE(pci,...)`, and should be set to `NULL` if you want to call the `probe()` function to check all **PCI** devices the system knows about.
`probe`	Points to a probing function which looks for your device.
`remove`	Points to a function that can be called whenever the device is removed, either by de-registration or by yanking out of a hot-pluggable slot.
`suspend`	Used by power management when a device goes to sleep.
`resume`	Used by power management when a device wakes up.

We won't give a detailed description of the functions pointed to in the above jump table. Functions which are unnecessary for a particular device can be left as `NULL`.

The ID table is an array of structures of `pci_device_id` which must end with a `NULL` entry:

```
2.6.31:  17 struct pci_device_id {
2.6.31:  18     __u32 vendor, device;           /* Vendor and device ID or PCI_ANY_ID*/
2.6.31:  19     __u32 subvendor, subdevice;     /* Subsystem ID's or PCI_ANY_ID */
2.6.31:  20     __u32 class, class_mask;        /* (class,subclass,prog-if) triplet */
2.6.31:  21     kernel_ulong_t driver_data;     /* Data private to the driver */
2.6.31:  22 };
```

This table is usually filled out with use of the `PCI_DEVICE()` macro, as in:

```
static struct pci_device_id tg3_pci_tbl[] = {
   {PCI_DEVICE(PCI_VENDOR_ID_BROADCOM, PCI_DEVICE_ID_TIGON3_5700)},
   {PCI_DEVICE(PCI_VENDOR_ID_BROADCOM, PCI_DEVICE_ID_TIGON3_5701)},
   ....
   {PCI_DEVICE(PCI_VENDOR_ID_ALTIMA, PCI_DEVICE_ID_ALTIMA_AC9100)},
   {PCI_DEVICE(PCI_VENDOR_ID_APPLE, PCI_DEVICE_ID_APPLE_TIGON3)},
       {}
};

MODULE_DEVICE_TABLE(pci, tg3_pci_tbl);
```

It is important to enable your device after you find it, before you do anything with it, by calling the function `pci_enable_device()`. This switches on the I/O and memory regions, allocates any missing resources that might be needed, and wakes up the device if was in suspended state. Usually this function would be called from the **probe** callback function.

The kernel contains an excellent manual on writing **PCI** device drivers in the file **/usr/src/linux/Documentation/pci.txt**.

- A **PCI** device may be of many different kinds, such as a network, character, or block device. In addition to registering as a **PCI** device, it will also have to register as the particular kind of device it is.
- This is normally done in the `probe()` callback function, where one would call functions such as `register_netdev()`, etc.
- Likewise, the `shutdown()` callback function would deregister the device with functions such as `unregister_netdev()`.

22.3 PCI Structures and Functions

The header file **/usr/src/linux/include/linux/pci.h** defines symbolic names for numeric values used by PCI functions, for register locations and values. The header file **/usr/src/linux/include /linux/pci_ids.h** has device and vendor specific definitions and is included from `pci.h`.

The basic structure describing a PCI device is of the type `pci_dev` defined in **/usr/src/linux/include /linux/pci.h**. This is a long structure and we don't need to get into the details here.

Locating devices can be done with:

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

struct pci_dev *pci_get_device (unsigned int vendor, unsigned int device,
                                struct pci_dev *from);
struct pci_dev *pci_get_device_reverse (unsigned int vendor, unsigned int device,
                                struct pci_dev *from);
struct pci_dev *pci_get_class (unsigned int class, struct pci_dev *from);
```

and some other related functions.

The function `pci_get_device()` requests information about the device. If `vendor` and/or `device` is specified as `PCI_ANY_ID=-1`, all devices are matched. Before initializing a chain of devices, the value of `from` should be set to `NULL`. These functions are often called in a loop. If `from` is set to `NULL` at the beginning, it will return `NULL` at the end.

If the return value of these functions is not `NULL`, the data structure describing the device is returned, and the reference count for the device is incremented. When the device is released (say on module unloading) one must call the function:

```
void pci_dev_put(struct pci_dev *dev);
```

to decrement the reference count; otherwise it can't be removed from the system if it is hotplug-able.

Remember that after finding the device, before you do anything with it, you need to call the function `pci_enable_device (struct pci_dev *dev)` in order to enable it.

Example:

To locate and enable one particular device:

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

struct pci_dev *pdev = NULL;

if ( !(pdev = pci_get_device(PCI_VENDOR_ID_INTERG, PCI_DEVICE_ID_INTERG_1682, pdev))
      return -ENODEV;
pci_enable_device(pdev);
```

If desired, the function

```
char *pci_name(struct pci_dev *pdev)'
```

can be called to get the bus, device and function numbers. It doesn't do this by probing hardware, but instead by traversing the known list of present devices.

One can also use the macro:

```
struct pci_dev *pdev;
for_each_pci_dev(pdev){ .... }
```

to step through all **PCI** devices.

22.4 Accessing Configuration Space

The configuration space can be accessed through 8-bit, 16-bit, or 32-bit data transfers.

```
int pci_read_config_byte   (struct pci_dev *dev, u8 where, u8 *val);
int pci_read_config_word   (struct pci_dev *dev, u8 where, u16 *val);
int pci_read_config_dword  (struct pci_dev *dev, u8 where, u32 *val);
int pci_write_config_byte  (struct pci_dev *dev, u8 where, u8 *val);
int pci_write_config_word  (struct pci_dev *dev, u8 where, u16 *val);
int pci_write_config_dword (struct pci_dev *dev, u8 where, u32 *val);
```

These functions read from or write to `val`, to or from the configuration space of the device, identified by `dev`. The byte offset from the beginning of the configuration space is given by `where`.

Note the use of the types `u8, u16, u32, u64`. These are kernel unsigned data types to be used when you must be exactly sure of the length in bits. (There are also signed types, `s8, s16, s32, s64` which are rarely used.) You can also use these from user-space as long as you prefix them with a double underscore (e.g., `__u32`) and include `linux/types.h`.

Multi-byte entries in the configuration registers are in **little-endian** order, according to the **PCI** standard, which is also the convention on **x86** platforms. The above data types and functions handle any byte ordering that needs to be done transparently, when one is on a system like the **SPARC**, which is **big-endian**, so you don't have to worry about bit order. But you should be aware of it. Byte ordering is taking care of for word and dword functions. (Note some architectures, such as **alpha** and **IA64** are actually **bi-endian** and can be configured either way.)

Configuration variables are best accessed using the symbolic names defined in **/usr/src/linux /include/linux/pci_regs.h**, e.g.

```
pci_read_config_byte (dev, PCI_REVISION_ID, &revision);
```

It is also possible to use the **setpci** utility to get and set values in the configuration register.

22.5 Accessing I/O and Memory Spaces

As mentioned, one will have to access not only the configuration registers, but also I/O ports and memory regions associated with **PCI** devices. While it is possible to go hunting for these resources in the configuration registers, it is easier to use the generic resource management functions provided by the kernel.

The relevant functions are:

```
unsigned long pci_resource_start (struct pci_dev *dev, int bar);
unsigned long pci_resource_end   (struct pci_dev *dev, int bar);
unsigned long pci_resource_len   (struct pci_dev *dev, int bar);
unsigned long pci_resource_flags (struct pci_dev *dev, int bar);
```

in which **bar** stands for Base Address Register.

The first two functions return the starting and ending address of one of the up to 6 I/O regions that can be found on the device; the parameter bar thus ranges from 0 to 5 and selects which one is requested.

The last function returns the flags associated with the device, which are defined in **/usr/src/linux/include/linux/ioport.h**.

Here's an example of usage from **/usr/src/linux/drivers/net/8139too.c**:

```
2.6.31: 738 static __devinit struct net_device * rtl8139_init_board (struct pci_dev *pdev)
2.6.31: 739 {
...
2.6.31: 764     rc = pci_enable_device (pdev);
2.6.31: 765     if (rc)
2.6.31: 766             goto err_out;
2.6.31: 767
2.6.31: 768     pio_start = pci_resource_start (pdev, 0);
2.6.31: 769     pio_end = pci_resource_end (pdev, 0);
2.6.31: 770     pio_flags = pci_resource_flags (pdev, 0);
2.6.31: 771     pio_len = pci_resource_len (pdev, 0);
2.6.31: 772
2.6.31: 773     mmio_start = pci_resource_start (pdev, 1);
2.6.31: 774     mmio_end = pci_resource_end (pdev, 1);
2.6.31: 775     mmio_flags = pci_resource_flags (pdev, 1);
2.6.31: 776     mmio_len = pci_resource_len (pdev, 1);
...
```


22.6 PCI Express

PCI was introduced in 1991 and despite some enhancements such as **PCI-X**, it has shown its age. In particular, it's bandwidth is limited to 133 MB/s. Furthermore this bandwidth is shared among all the devices on the bus, and competition for it must be negotiated.

In 1997 a separate **AGP** (**A**ccelerated **G**raphics **P**ort) was added with its own dedicated bandwidth. But **AGP** has now disappeared in recent motherboards.

PCI Express (usually denoted as **PCIe**) was introduced in 2004 and is gradually taking over. Its main quality is that it is a **point-to-point** connection; bandwidth is not shared, communication is direct via a switch that directs data flow. Furthermore, hot plugging devices is far easier, and less power is consumed than for **PCI**.

Each device communicates through a number of serial **lanes** each of which is bi-directional and has a 250 MB/s rate in each direction for a possible 500 MB/s total data transfer rate.

The number of lanes depends on the kind of slot; there are 1, 2, and 16 lane slots available; the x16 slot, for example, can accommodate up to 8000 MB/s and is used by graphic cards. x32 and x64 lane cards and slots are also in the standard.

Any **PCIe** card will fit and work correctly in any slot that is as least as large as it is; e.g., you can put an x4 card in an x16 slot, it will just use fewer lanes.

Device drivers written for **PCI** will still work for **PCIe** as the standard was designed to cause as little disruption as possible.

22.7 Labs

Lab 1: PCI Utilities

The **pciutils** package (**http://mj.ucw.cz/pciutils.html**) contains the following utilities:

- **lspci** displays information about **PCI** buses and connected devices, with many options.
- **setpci** can interrogate and configure properties of **PCI** devices.
- **update-pciids** obtains the most recent copy of the **PCI ID** database and installs it on your system.

Run **update-pciids** to update your database. If it fails because the URL pointed to in the script is down or obsolete try obtaining it directly from **http://pci-ids.ucw.cz/**. The location of the downloaded file (pci.ids) depends on your distribution, but will be somewhere under /usr/share. (Entering `locate pci.ids` will tell you.)

Get more than basic information from **lspci**. You can get details from `man lspci` or `lspci -help`. For example, to get very verbose information about all **Intel** devices on your system you could type `lspci -vv -d 0x8086:*`, or for **AMD** devices, `lspci -vv -d 0x1022:*`. Experiment with the `-x(xx)` options to get detailed dumps of the configuration registers.

Use `setpci` to evaluate or change specific values in the configuration register. For example you could find out the device identifier for all **Intel** devices on your system with `setpci -vD -d 0x1022:* DEVICE_ID`, where the `-D` option prevents actual changes from happening. See the **man** pages for examples of changing various configuration register entries.

Lab 2: PCI Devices

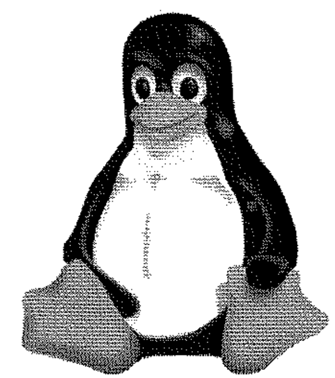
Write a module that scans your **PCI** devices, and gathers information about them.

For each found device, read some information from its configuration register. (Make sure you read **/usr/src/linux/include/linux/pci_regs.h** and **/usr/src/linux/include/linux/pci_ids.h** to get symbolic names.) Fields you may wish to obtain could include: `PCI_VENDOR_ID`, `PCI_DEVICE_ID`, `PCI_REVISION_ID`, `PCI_INTERRUPT_LINE`, `PCI_LATENCY_TIMER`, `PCI_COMMAND`.

The information you obtain should agree with that obtained from **lspci**.

Chapter 23

Direct Memory Access (DMA)



We'll learn about **DMA** under **Linux**. We'll consider how **DMA** uses interrupts for synchronous and asynchronous transfers, how **DMA** buffers must be allocated, and virtual to physical (and bus) address translation. Then we'll look in some detail how **DMA** is deployed for the **PCI** bus, considering both **consistent** and **streaming** transfers, and the use of **DMA Pools**. We'll examine gather/scatter mappings. Finally we'll consider **DMA** for the **ISA** bus.

23.1 What is DMA?

Direct Memory Access (**DMA**) permits peripheral devices to transfer data to or from system memory while bypassing CPU control. Proper use of **DMA** can lead to dramatic performance enhancement. Most non-trivial peripherals are likely to have **DMA** capabilities.

The specifics of **DMA** transfers are very hardware-dependent, both in the sense of the CPU involved (e.g., **x86** or **Alpha**), and the type of data bus (e.g., **PCI**, **ISA**, etc.), and to some degree these degrees of freedom are independent.

However, since the 2.4 kernel series the goal has been to present a unified, hardware-independent interface. This was achieved in the 2.6 kernel series, permitting one to deal with more abstract methods rather than getting deep into the hardware particularities..

On the **x86** platform, **DMA** operates quite differently for **ISA** and **PCI** devices. One could say:

- **ISA:** The hardware is relatively less complex, but the device drivers are more complicated and have to work hard to manage **DMA** transfers.
- **PCI:** The hardware is more complex, but the device drivers are less complicated and have an easier time managing **DMA** transfers.

We'll concentrate on the **PCI** bus which is more modern and most widespread.

23.2 DMA and Interrupts

The efficiency of **DMA** transfers is very dependent on proper interrupt handling. Interrupts may be raised when the device acquires data, and are always issued when the data transfer is complete.

Transfers require a **DMA**-suitable buffer, which must be contiguous and lie within an address range the device can reach, and we will discuss how such buffers can be allocated and released. In the following we will assume that either: such a buffer exists before the transfer and is not released but will be re-used in subsequent transfers; or must be allocated before the transfer begins and released when it is complete.

Transfers can be triggered **synchronously**, or directly, such as when an application requests or *pulls* data through a `read()`, in which case:

- The hardware is told to begin sending data
- The calling process is put to sleep.
- The hardware puts data in the **DMA** buffer.
- The hardware issues an interrupt when it is finished.
- The interrupt handler deals with the interrupt, acquires the data, and awakens the process, which can now read the data.

When an application pushes (or writes) data to the hardware one also has a synchronous transfer and the steps are similar.

Transfers can also be triggered **asynchronously** when the hardware acquires and *pushes* data to the system even when there are no readers at present. In this case:

- The driver must keep a buffer to warehouse the data until a `read()` call is issued by an application.
- The hardware announces the arrival of data by raising an interrupt.
- The interrupt handler tells the hardware where to send the data.
- The peripheral device puts the data in the **DMA** buffer.
- The hardware issues an interrupt when it is finished.
- The interrupt handler deals with the data, and awakens any waiting processes.

Note that while pushes and pulls have many similar steps, the asynchronous transfer involves two interrupts per transfer, not one.

23.3 DMA Memory Constraints

DMA buffers must occupy *contiguous* memory; Thus you can't use `vmalloc()`, only `kmalloc()` and the `__get_free_pages()` functions. Note you can use also use the abstracted allocation functions we will detail shortly.

If you specify `GFP_DMA` as the priority the physical memory will not only be contiguous, on **x86** it will also fall under `MAX_DMA_ADDRESS=16 MB`.

For **PCI** this should be unnecessary and wasteful, but there exist **PCI** devices which still have addressing limitations (sometimes because they were poorly crafted from an **ISA** device.) Thus it is actually necessary to check what addresses are suitable.

Because the hardware is connected to a peripheral bus which uses **bus addresses** (while both kernel and user code use virtual addresses) conversion functions are needed. These are used when communicating with the Memory Management Unit (MMU) or other hardware connected to the CPU's address lines:

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

unsigned long virt_to_bus  (volatile void *address);
void *bus_to_virt  (unsigned long address);

unsigned long virt_to_phys (volatile void *address);
void *phys_to_virt (unsigned long address);
```

You can look at the header file to see how these macros are defined.

On the **x86** platform bus and physical addresses are the same so these functions do the same thing.

23.4 DMA Directly to User

High-bandwidth hardware (e.g., a video camera) can obtain lots of speed-up by going straight to the user; i.e., without using **DMA** to first get the data to kernel-space and then transferring to user-space. If one wants to do this by hand it is tricky; the steps are:

- Lock down the user pages.
- Set up a **DMA** transfer for each page.
- When the **DMA** is done, unlock the pages.

If these steps seem familiar, it is because they are essentially what the `get_user_pages()` **API** does for you; you'll of course still have to do the **DMA** transfers properly.

23.5 DMA under PCI

The **API** used for **DMA** is platform-independent, and involves a generic structure of type `device`, which may or may not be **PCI** in nature. This structure is embedded in the `pci_dev` structure, so to get at it you'll have to also include **/usr/src/linux/include/linux/pci.h**.

If one has a device with addressing limitations, the first thing to do is to check whether **DMA** transfers to the desired addresses are possible, with:

```
#include <linux/dma-mapping.h>

int dma_supported (struct device *dev, u64 mask);
```

For example, if you have a device that can handle only 24-bit addresses, one could do:

```
struct pci_dev *pdev;
if ( dma_supported (pdev->dev, 0xffffff) ){
    pdev->dma_mask = 0xffffff;
} else {
    printk (KERN_WARNING "DMA not supported for the device\n");
    goto device_unsupported;
}
```

If the device supports normal 32-bit operations, one need not call `dma_supported()` or set the mask.

In order to set up a **DMA** transfer one has to make a **DMA Mapping**, which involves two steps; allocating a buffer, and generating an address for it that can be used by the device. The details of how this is done are architecture dependent, but the functions for allocating and freeing are the same across platforms:

```
void *dma_alloc_coherent (struct device *dev, size_t size, dma_addr_t *dma_handle, gfp_t flag);
void dma_free_coherent (struct device *dev, size_t size, void *vaddr, dma_addr_t dma_handle);
```


The allocation function returns a kernel virtual address for the buffer, of length `size` bytes. The third argument points to the associated address on the bus (which is meant to be used opaquely.) The `flag` argument controls how the memory is allocated, and is usually `GFP_KERNEL` or `GFP_ATOMIC` if sleeping is not allowed such as when in interrupt context. If the mask requires it, `GFP_DMA` can also be specified. This memory can be freed with `dma_free_coherent()` which requires both addresses as arguments.

Memory regions supplied with `dma_alloc_coherent()` are used for so-called **Coherent DMA Mappings**, which can also be considered as **synchronous** or **consistent**. These have the following properties:

- The buffer can be accessed in parallel by both the CPU and the device.
- A write by either the device or the CPU can immediately be read by either, without worrying about cache problems, or flushing. (However, you may still need to use the various memory barriers functions, as the CPU may reorder I/O instructions to consistent memory just as it does for normal system memory.)
- The minimum allocation is generally a page. In fact on **x86** one actually always obtains a number of pages that is a power of 2, so it may be expensive.

Since this method is relatively expensive, it is generally used for **DMA** buffers that persist through the life of the device. A good example of its use would be for network card **DMA** ring descriptors.

For single operations, one sets up so-called **Streaming DMA mappings**, which can also be considered as **asynchronous**. These are controlled with

```
dma_addr_t dma_map_single (struct device *dev, void *ptr, size_t size,
                           enum dma_data_direction direction);
void dma_unmap_single (struct device *dev, dma_addr_t dma_addr, size_t size,
                       enum dma_data_direction direction);
```

A pointer to a previously allocated memory region is passed through the `ptr` argument; this must be allocated in **DMA**-suitable fashion; i.e., contiguous and in the right address range. The `direction` argument can have the following values:

Table 23.1: **DMA transfer direction values**

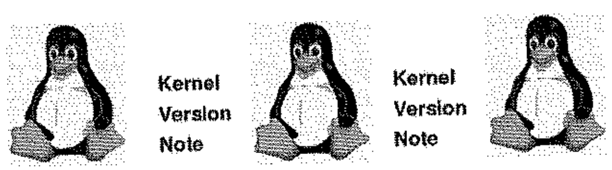
Value	Meaning
PCI_DMA_TODEVICE	Data going to the device, e.g., a write.
PCI_DMA_FROMDEVICE	Data coming from the device, e.g., a read.
PCI_DMA_BIDIRECTIONAL	Data going either way.
PCI_DMA_NONE	Used for debugging; any attempt to use the memory causes a crash.

Streaming **DMA** mappings might be used for network packets, or filesystem buffers. They have the following properties:

- The direction of a transfer must match the value given during the mapping.
- After a buffer is mapped, it belongs to the **device**, not the CPU; the driver should not touch the buffer until it has been unmapped.
- Thus for a write the data should be placed in the buffer before the mapping; for a read it should not be touched until after the unmapping (which could be done after the device signals, through an interrupt, that it is through with the transfer.)

A third kind of mapping is a so-called **Scatter-gather DMA Mapping**. This permits several buffers, which may be non-contiguous, to be transferred to or from the device at one time.

It is also possible to set up a **DMA pool**, which works pretty much like a memory cache. We'll consider that next.



- Rather than using a generic interface, the 2.4 kernel used a **PCI**-specific interface. The main functions are in one-to-one correspondence with the generic ones and are:

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

int pci_dma_supported (struct pci_dev *dev, u64 mask);
int pci_set_dma_mask (struct pci_dev *dev, u64 mask);
void *pci_alloc_consistent (struct pci_dev *dev, size_t size,
                            dma_addr_t *dma_handle);
void pci_free_consistent (struct pci_dev *dev, size_t size, void *vaddr,
                            dma_addr_t dma_handle);
dma_addr_t pci_map_single (struct pci_dev *dev, void *ptr, size_t size,
                            int direction);
void pci_unmap_single (struct pci_dev *dev, dma_addr_t dma_addr, size_t size,
                            int direction);
```

- While this older **API** has not been removed, it is now just a wrapper around the more general interface, which should be used in any new code.

23.6 DMA Pools

Suppose you need frequent small **DMA** transfers. For coherent transfers, `dma_alloc_coherent()` has a minimum size of one page. Thus, a good choice would be set up a **DMA Pool**, which is essentially a slab cache intended for use in **DMA** transfers.

The basic functions are:

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

struct dma_pool *dma_pool_create (const char *name, struct device *dev, size_t size,
                                  size_t align, size_t allocation);
void dma_pool_destroy (struct dma_pool *pool);
void *dma_pool_alloc (struct dma_pool *pool, gfp_t mem_flags, dma_addr_t *handle);
void dma_pool_free (struct dma_pool *pool, void *vaddr, dma_addr_t addr);
```

No actual memory is allocated by `dma_pool_create()`; it sets up a pool with a name pointed to by `name`, to be associated with the device structure pointed to by `dev`, of `size` bytes.

The `align` argument (given in bytes) is the hardware alignment for pool allocations. The final argument, `allocation`, if non-zero specifies a memory boundary allocations should not cross. For example, if `allocation=PAGE_SIZE`, buffers in the pool will not cross page boundaries.

The actual allocation of memory is done with `dma_pool_alloc()`. The `mem_flags` argument gives the usual memory allocation flags (`GFP_KERNEL`, `GFP_ATOMIC`, etc.) The return value is the kernel virtual address of the **DMA** buffer, which is stored in `handle` as a bus address.

To avoid memory leaks, buffers should be returned to the pool with `dma_pool_free()`, and when all have been released the pool can be wiped out with `dma_pool_destroy()`.

Note that the memory allocated with the use of the pool will have consistent **DMA** mappings, which means both the device and the driver can use it without using cache flushing primitives.

23.7 Scatter/Gather Mappings

It is easiest to do a **DMA** transfer if you have only one (large or small) contiguous buffer to work with. Then you can just give a starting address and a length and get the transfer in motion.

However, one often might have several buffers requiring transfer at the same time, and they might not be physically contiguous. This might occur due to:

- A `readv()` or `writev()` system call.
- A disk I/O request.
- Transfer of a list of pages in a mapped kernel I/O buffer (such as one might have when using `get_user_pages()`.)

Of course one can chain together a series of individual requests, each one of which represents a contiguous region. But many devices are capable of assisting at the hardware level; a so-called

scatterlist of pointers and lengths can be given to the device and then it will take care of doing it all as one operation.

In order to accomplish this you first have to set up an array of structures describing the buffers requiring transfer. For **x86** this structure is described in **/usr/src/linux/arch/x86/include /asm/scatterlist.h** and looks like:

```
struct scatterlist {
    struct page         *page;
    unsigned int        offset;
    dma_addr_t          dma_address;
    unsigned int        length;
};
```

The driver sets the `page`, `offset`, and `length` fields for each buffer in the array; Note `length` is specified in bytes.

The `dma_address` field will be filled in by the function:

```
int dma_map_sg (struct device *dev, struct scatterlist *sg,
                int nents, enum dma_data_direction direction);
```

where `nents` is the number of buffers in the array. This function returns the number of buffers to transfer. This can be less than `nents` because any physically adjacent buffers will be combined.

Kernel Version Note

Kernel Version Note

In kernel version 2.6.24, the `scatterlist` structure was changed to:

```
struct scatterlist {
    unsigned long  page_link;
    unsigned int   offset;
    unsigned int   length;
    dma_addr_t     dma_address;
    unsigned int   dma_length;
};
```

Filling in the fields is best done with:

```
void sg_set_page(struct scatterlist *sg, struct page *page, unsigned int len,
                 unsigned int offset);
```

defined in **/usr/src/linux/include/linux/scatterlist.h** together with a lot of other useful convenience functions for accessing gather-scatter structures.

Once this is done it is time to transfer each buffer. Because of architectural differences, one should not refer directly to the elements of the `scatterlist` data structure, but instead use the macros:

```
dma_addr_t sg_dma_address (struct scatterlist *sg);
unsigned int sg_dma_len (struct scatterlist *sg);
```

which return the bus (**DMA**) address and length of the buffer (which may be different than what was passed to `dma_map_sg()` because of buffer coalescence.)

After the full transfer has been made, one calls

```
int dma_unmap_sg (struct device *dev, struct scatterlist *sg, int nents,
                  enum dma_data_direction direction);
```

where `nents` is the original value passed to the mapping function, not the coalesced value.

23.8 DMA under ISA

There are two kinds of **ISA** transfers using **DMA**: *native* transfers using standard motherboard **DMA**-controller circuitry, and *ISA-busmaster* hardware, where the peripheral device controls everything. This latter type is rare and we won't discuss it, but it is similar to **PCI** transfers.

The **DMAC** (8237 DMA Controller) maintains information such as the direction and size of the transfer, the memory address, and the status of ongoing transfers. When a **DMA** request signal is received by the **DMAC**, it drives the signal lines so the device can read and write data. These circuits are now part of the motherboard chipset, rather than separate 8237 chips.

The peripheral device must send a **DMA** request signal when it is ready for a transfer. It raises an interrupt when the transfer is done.

The device driver tells the **DMAC** the direction, address and size of the transfer, tells the peripheral to get ready, and answers the interrupt issued when the transfer completes.

In all but the oldest PC's, there are two **DMAC**'s, and each has four *channels*, each of which is associated with a set of **DMA** registers.

The second (master) controller is connected to the CPU; the first (slave) controller is connected to channel 0 of the master controller.

Channels 0 through 3 on the slave are 8-bit channels; channel 4 (the first channel on the master) is used to *cascade* the slave controller. Channels 5 through 7 on the master are the 16-bit channels.

The maximum size of an 8-bit transfer is 64 KB; for 16-bits it is 128 KB. (It is stored as a 16-bit number.)

DMA usage is requested and freed with the following functions:

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

int request_dma (unsigned int dmanr, const char *device_id);
void free_dma (unsigned int dmanr);
```

where dmanr must be less than MAX_DMA_CHANNELS (i.e., 0 through 7), and device_id is the name which appears in **/proc/dma**.

An **IRQ** line is always needed when using a **DMA** device. The **DMA** channel should be requested *after* the **IRQ** and released before it.

An Example:

```
/* in initialization */

   request_irq(my_irq, my_interrupt, IRQF_DISABLED, 'my_dma', NULL) ;
   request_dma(my_dmanr, 'my_dma');

/* in cleanup */

   free_dma( my_dmanr );
   free_irq( my_irq, NULL );
```

There are a number of other functions used to communicate with the **DMAC**, all of which are coded in asm/dma.h and kernel/dma.c. It is hard to give generic examples for **DMA** as it is very device dependent, but one should look at various kernel drivers for more information.

23.9 Labs

Lab 1: DMA Memory Allocation

Write a module that allocates and maps a suitable **DMA** buffer, and obtains the bus address handle.

Do this in three ways:

- Using dma_alloc_coherent().
- Using dma_map_single()
- Using a **DMA Pool**.

You can use NULL for the device and/or pci_dev structure arguments since we don't actually have a physical device.

Compare the resulting kernel and bus addresses; how do they differ? Compare with the value of PAGE_OFFSET.

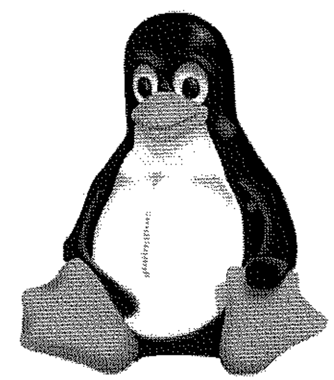
In each case copy a string into the buffer and make sure it can be read back properly.

In the case of dma_map_single(), you may want to compare the use of different direction arguments.

We give two solutions, one with the bus-independent interface, and one with the older **PCI API**.

Chapter 24

Network Drivers I: Basics



We'll consider the layered approach to networking found in **Linux**, and how network drivers differ from character and block device drivers. We'll explain how network drivers are loaded, unloaded, opened and closed.

24.1 Network Layers and Data Encapsulation

Networking applications communicate with servers and clients which are also networking applications. These applications (or daemons) may either be on remote hosts or on the local machine.

For the most part these applications are constructed to be independent of the actual hardware, type of network involved, routing, and specific protocols involved. This is not a general rule, however, as sometimes the application may work at a lower level or require certain features.

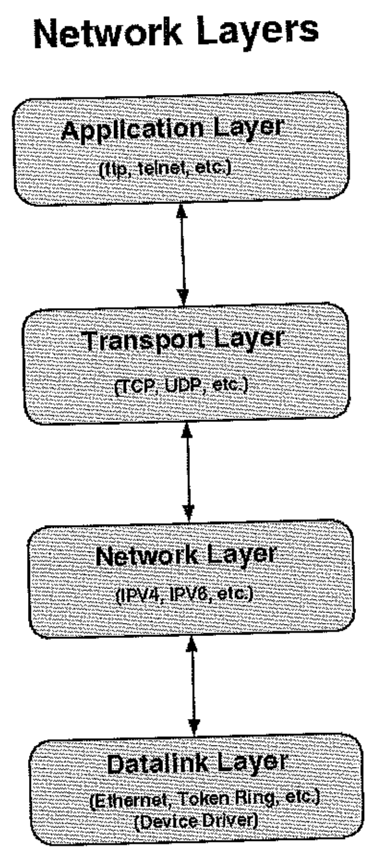


Figure 24.1: **Network layers**

Thus networking can always be seen as consisting of a number of stacked layers. This stack may be categorized in a number of ways, but a simple description appropriate for **Linux** (in which traffic moves both ways) would be:

- **Application Layer:** This can be familiar programs such as **ftp, http** and **smtp**, that communicate across the Internet, or programs such as **X** clients and servers that communicate either across the Internet or on the local machine, as well as any custom network application.
- **Transport Layer:** This is the method of data encapsulation, error checking protocols, etc. The two most common examples are: **streaming** (usually **TCP** (Transmission Control Protocol)) connection-oriented; and **datagram** (usually **UDP** (User Datagram Protocol)) connectionless. Other examples include **SLIP** and **PPP**.
- **Network Layer:** This describes how data is to be sent across the network, containing routing information etc. The most common protocols are **IPV4** (Internet Protocol Version 4) or the newer **IPV6**, and **Unix** for local machine communication. Other examples include **ARP** and **ICMP**
- **Datalink Layer:** This is the hardware part. It includes both the type of device (**Ethernet, Token Ring**, etc.) and the actual device driver for the network card.

Networking applications send and receive information by creating and connecting to **sockets**, whose endpoints may be anywhere. Data is sent to and received from sockets as a **stream**. This is true whether or not the underlying transport layer deals with connectionless un-sequenced data (such as **UDP**), or connection-oriented sequenced data (such as **TCP**). Note that various combinations of these layers are possible, such as **TCP/IP** or **UDP/IP**.

The basic data unit that moves through the networking layers and through the network is the **packet**, which contains the data but also has headers and footers containing control information. Within the kernel, the packet is described by a **socket buffer**, a data structure of type **sk_buff**.

These headers and footers contain information such as the source and destination of the packet, various options about priority, sequencing, and routing, identification of the device driver associated with the socket, etc.

When an application writes into a socket, the transport layer creates a series of one or more packets and adds control information. It then hands them off to the network layer which adds more control information, and decides where the packets are going; Packets going out on the network are handed off to the datalink layer and the device driver.

When packets of data are received by the datalink layer, it first sees if they are intended for the local machine, and if so it processes them and hands them off to the network layer, which then passes them through to the transport layer, which sequences the packets, and finally strips out the data and sends it back to the application.

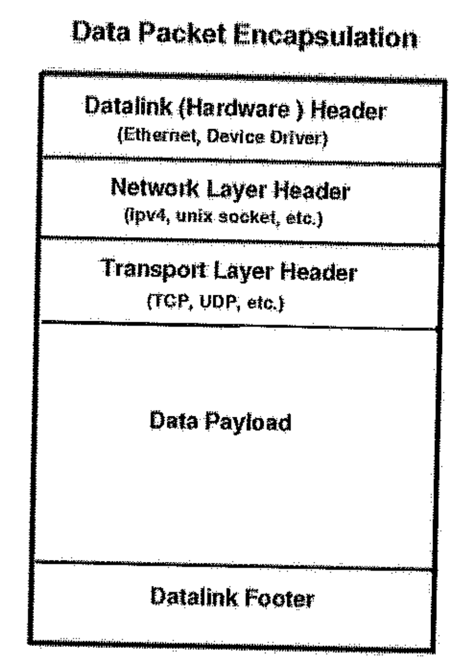


Figure 24.2: **Data Packet Encapsulation**

As packets move up and down through the networking layers, the kernel avoids repeated copying by passing pointers to the encapsulated data buffer, or payload, in the packet, while modifying headers and footers as necessary.

The outermost header and footer correspond to the **datalink** layer, which describes the hardware, device driver, and the hardware type of network, such as **Ethernet** or **Token Ring**. For instance, the **MAC** address will appear here.

The next layer describes the network protocol, such as **IPV4**, and the innermost header describes the transmission protocol, such as **TCP** or **UDP**.

For reasons of optimization, the network and transport layer are not as completely separated as they might be in principle. For instance the network layer implementations may each contain implementations of the transport layer, rather than passing always through a common code base. For instance, both **Internet** and local **Unix** sockets have their own datagram (connectionless) code.

24.2 Datalink Layer

Applications have no knowledge of the hardware through which the computer will be connected to the network, except for specialized programs used for monitoring and diagnostic purposes.

For instance the computer may be connected via **Ethernet**, **Bluetooth** or **Token Ring**, each of which has its own kind of network interface card (**NIC**), or through a **PPP** connection through a serial modem. Even within a given hardware type there are many choices; **Linux** supports probably hundreds of different kinds of **Ethernet** NIC's.

Thus there are two hardware components that the kernel needs to deal with; the datalink layer (**Ethernet** etc.) and the device driver which handles the NIC. These are not completely independent degrees of freedom as a given NIC will work only with a given datalink layer; thus the device drivers are always associated with a given datalink layer.

Generally speaking, applications are independent of both these layers as well as the network and transport layers.

24.3 Network Device Drivers

We are going to concentrate on the networking **datalink** layer, i.e., the device driver. We are not going to consider the application layer or the network and transport layers except for how they interact with the device drivers.

As for other classes of devices, network drivers can either be built-in or modular.

Network device drivers are fundamentally different than **character** and **block** device drivers and have no associated filesystem node. Instead they are identified by a name, such as `eth0` or `ppp0`.

At the kernel level they work with **packet transmission** and **reception**, not **read** and **write** operations. A network driver **asks to push** incoming packets; other drivers are **asked to send** a buffer towards the kernel.

To make writing a network device driver easier, templates are included in the kernel sources. For **ISA** hardware, the relevant file is **/usr/src/linux/drivers/net/isa-skeleton.c**, and for **PCI** it is **/usr/src/linux/drivers/net/pci-skeleton.c**.

24.4 Loading/Unloading

Network drivers are loaded and unloaded with:

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

int  register_netdev   (struct net_device *);
void unregister_netdev (struct net_device *);
```

Usually one registers the device in the initialization callback function and unregisters it in the exit callback.

The command

```
ifconfig interface [aftype] options  [address]
```

can be used to start, stop, and configure interfaces.

`register_netdev()` function invokes a specified initialization routine to probe for the network card, and fills in the `net_device` structure, which includes the interface name (which can be dynamically assigned.) `unregister_netdev()` removes the interface from the list of interfaces. Any memory associated with it should be freed.

The `net_device` structure contains all information about the device, and we will discuss it in detail. To obtain proper reference counting it must be allocated and freed dynamically with:

```
struct net_device *alloc_netdev (int sizeof_priv, const char *name,
                                 void (*setup)(struct net_device *));
void free_netdev (struct net_device *dev);
```

where:

- `sizeof_priv` is the size of the `priv` private data field in the `net_device` data structure.
- `name` is the name of the device; if a format such as `"mynet%d"` is chosen the name will be filled out dynamically as devices are brought up (`mynet0, mynet1, ...`).
- `setup()` points to an initialization function that will set up remaining fields in the data structure. For a standard network device the function `ether_setup()` can be used, or it can be called from a supplied function.

For convenience one can call the simpler function:

```
struct net_device *alloc_etherdev (int sizeof_priv);
```

which supplies `"eth%d"` for the name and points to `ether_setup()` for initialization.

One should never access the `priv` field directly as to do so would inhibit performance, flexibility and reference counting. Instead one uses the inline function `netdev_priv()` as in:

```
struct my_priv *priv = netdev_priv(dev);
```

One has no need to, and should never do `kfree (dev->priv)`, as the private area is allocated along with the entire structure, and thus is released at the same time.

It is important to initialize the device properly. First your **setup()** function will get called and then it can call `ether_setup()` which gives standard values for an Ethernet device. Then specific elements in the structure can be directly initialized. For example:

```
void mynet_setup (struct net_device *dev){
      ....
      ether_setup (dev);
      dev->open           = mynet_open;
      dev->stop           = mynet_close;
      dev->hard_start_xmit = mynet_xmit;
      dev->tx_timeout     = mynet_timeout;
      dev->do_ioctl       = mynet_ioctl;
      dev->get_stats      = mynet_getstats;
      dev->watchdog_timeo  = timeout;
}
```

There may be other functions that you may need to point to as well, if you don't want the defaults installed by `ether_setup()`.

While the `net_device` structure also contains an `int init()` function pointer which will be called (if it exists) by `register_netdevice()`, this is an older usage which has been retained but should not be used in new drivers..

Kernel Version Note Kernel Version Note

- Beginning with the 2.6.29 kernel many function pointers have been moved out of the `netdevice` structure into a structure of type `net_device_ops`. For the time being a compatibility layer has been maintained but will eventually be phased out as drivers are migrated over to the new layout. We will discuss this in the next section.

24.5 Opening and Closing

The opening and closing operations of a network device driver are similar to those for character and block devices, although how they are invoked from user-space is somewhat different. The callback functions are pointed to in the `net_device` data structure:


```
int (*open) (struct net_device *dev);
int (*stop) (struct net_device *dev);
```

As with the other sorts of drivers, it is generally a good idea not to allocate resources until they are needed; thus one shouldn't request interrupts, memory and other resources at device initialization, but rather do it when the device is first opened, or used. Likewise resources may be released upon closing, but it is often more efficient to let them remain until device unloading in case the network interface is reopened.

Remember that there is no filesystem entry point for a network device driver, and thus they are not opened by an `open()` call on a device node. Rather a program such as **ifconfig** will open and close the device by the use of `ioctl()` commands (on socket descriptors).

The opening process is generally done with sending two of these commands. The first uses the `SIOCSIFADDR` command to assign an address. The second sends `SIOCSIFFLAGS` to set the `IFF_UP` bit in `dev->flags` to denote turning the interface on.

The first command (setting the address) is handled entirely by the layers above the device driver. The second command (turning the device on) causes the invocation of the `open()` method for the device.

Similarly, shutting down the device (say with **ifconfig eth# down**) sends the `SIOCSIFFLAGS` command to clear the `IFF_UP` bit, and invokes the driver's `stop()` function.

One of the steps the `open()` method must perform is to get the hardware **MAC** address into `dev->dev_addr`, which has a length of `ETH_ALEN`.

Another step required is to start up the **transmit queue** for the device. There are a number of functions associated with this facility:

```
void netif_start_queue (struct net_device *dev);
void netif_stop_queue  (struct net_device *dev);
void netif_wake_queue  (struct net_device *dev);
```

During opening (or initialization) the function `netif_start_queue()` starts up the transmit queue. The `netif_stop_queue()` function can be used to mark the device as unable to transmit any more packets and should be used in the `stop()` method; the function `netif_wake_queue()` is used to resume such a temporary shutdown. There are some auxiliary functions which can be used to inquire about the state of the transmit queue; see **/usr/src/linux/include/linux/netdevice.h**.

We'll return to these functions when we discuss transmission.

24.6 Labs

Lab 1: Building a Basic Network Driver Stub

Write a basic network device driver.

It should register itself upon loading, and unregister upon removal.

Supply minimal `open()` and `stop()` methods.

You should be able to exercise it with:

```
insmod lab1_network.ko
ifconfig mynet0 up 192.168.3.197
ifconfig
```

Make sure your chosen address is not being used by anything else.

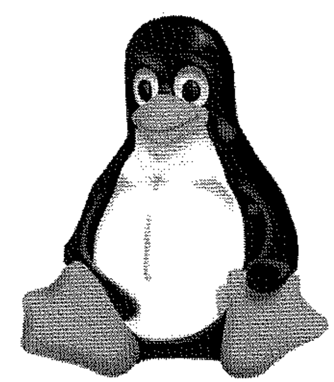
Warning: Depending on kernel version, your stub driver may crash if you try to bring it up or ping it. If you put in a trivial transmit function, such as

```
static int stub_start_xmit(struct sk_buff *skb,struct net_device *dev)
{
  dev_kfree_skb (skb);
  return 0;
}
```

this should avoid the problems.

Chapter 25

Network Drivers II: Data Structures



We'll consider the important **net_device** and **sk_buff** data structures, and the functions which manipulate socket buffers.

25.1 net_device Structure

The `net_device` structure is defined in **/usr/src/linux/include/linux/netdevice.h**. It is a large structure with many kinds of fields.

The first set of important entries includes:

Table 25.1: **Some important netdevice structure elements**

Field	Meaning
`char name[IFNAMSIZ]`	Name of the interface. If it contains a `%d` format string, the first available integer is appended to the base name (starting from 0). If the first character is blank or NULL, the interface is named `ethn`.
`unsigned long mem_end`	Shared (on-board) memory end.
`unsigned long mem_start`	Shared memory start. (Total on-board memory = `end-start`.)
`unsigned long base_addr`	Device I/O address. Assigned during device probe. (0 for probing, `0xFFe0` for no probing)
`unsigned char irq`	Device IRQ number
`unsigned char if_port`	On devices with multiports, specifies which port.
`unsigned char dma`	**DMA** channel allocated by the device (as on **ISA**.)
`unsigned long state`	Device state, including several flags.
`int features`	Tells the kernel about any special hardware capabilities possessed by the device.
`struct net_device *next`	Next device in the linked list of devices beginning at `dev_base`.
`int (*init)(struct net_device *dev)`	Device initialization function.

The rest of the structure has many different fields, most of which are assigned at device initialization. These describe device methods, interface information, and utility fields. We won't discuss all of these fields

The device methods section includes pointers to the device methods, or interface service routines. The most important are:

Table 25.2: **netdevice functional methods**

Field	Meaning
`int (*open) (struct net_device *dev);`	Open the interface, whenever **ifconfig** activates it. Should register resources (I/O Ports, IRQ, etc.), turn on hardware, etc. (fundamental)
`int (*stop) (struct net_device *dev);`	Stop the interface. Reverse the **open** operations. (fundamental)
`int (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev);`	Hardware start transmission. Send the packet in `sk_buff`. (fundamental)
`int (*poll) (struct net_device *dev, int *quota);`	Method for **NAPI**-compliant (interrupt-mitigated) drivers to operate in poll mode, with interrupts disabled. (optional)
`int (*hard_header) (struct sk_buff *skb, struct net_device *dev, unsigned short type, void *daddr, void *saddr, unsigned len);`	Build the hardware header from the source and destination hardware addresses. (fundamental)
`int (*rebuild_header) (struct sk_buff *skb)`	Rebuild the hardware header before packet is transmitted. (fundamental)
`void (*set_multicast_list) (struct net_device *dev);`	Called when multicast list for the device changes, or flags are set. (optional)
`int (*set_mac_address) (struct net_device *dev, void *addr);`	If the interface permits the hardware address to be changed. (optional)
`int (*do_ioctl) (struct net_device *dev, struct ifreq *ifr, int cmd);`	Perform interface-specific **ioctl** commands. (optional)
`int (*set_config) (struct net_device *dev, struct ifmap *map);`	Change the interface configuration. (fundamental)
`struct net_device_stats *(*get_stats)(struct net_device *dev);`	Gathers statistics for reporting, such as to **ifconfig**. (fundamental)
`struct iw_statistics *(*get_wireless_stats) (struct net_device *dev);`	Gathers statistics for wireless devices, such as to **ifconfig**. (fundamental)
`int (*change_mtu) (struct net_device *dev, int new_mtu);`	If there is a change in the MTU (Maximum Transfer Unit), do actions. (optional)
`void (*tx_timeout) (struct net_device *dev);`	Handle transmission (TX) timeouts. (fundamental)

int (*header_cache) (struct neighbour *neigh, struct hh_cache *hh);	Fills in the hh_cache structure with the results of an ARP query. (optional)
int (*header_cache_update) (struct hh_cache *hh, struct net_device *dev, unsigned char *haddr);	Updates destination address in the hh_cache structure. (optional)
int (*hard_header_parse) (struct sk_buff *skb, unsigned char *haddr);	Extracts the source address from the packet in skb, and puts it into the address pointed to by haddr. (optional)

A number of other fields describe **interface** information. Some of them pointing to initializing functions are:

Table 25.3: **netdevice interface information**

Field	Meaning
void ltalk_setup (struct net_device *dev);	Initialize fields for a **LocalTalk** device.
void fc_setup (struct net_device *dev);	Initialize fields for **fiber channel** devices.
void fddi_setup (struct net_device *dev);	Initialize fields for fiber distributed data interface (**FDDI**).
void hippi_setup (struct net_device *dev);	Initialize fields for a high-performance parallel interface (**HIPPI**) high speed interconnect driver.
void tr_configure (struct net_device *dev);	Initialize fields for **token ring** devices.

Some other interface fields that can be set directly, if you can't do it through one of the above functions are:

Table 25.4: **netdevice directly set fields**

Field	Meaning
unsigned short hard_header_len;	Hardware header length (number of bytes before the IP or other protocol header, 14 for Ethernet.)

unsigned mtu;	Maximum transfer unit. (Default for Ethernet is 1500.)
unsigned short type;	Interface hardware type. (For Ethernet is ARPHRD_ETHER.)
unsigned char addr_len;	MAC address length (6 for Ethernet.)
unsigned char broadcast[MAX_ADDR_LEN];	Hardware broadcast address (6 bytes of 0xff for Ethernet.)
unsigned char dev_addr[MAX_ADDR_LEN];	Hardware MAC address.
unsigned short flags;	Interface flags.

The flags entry is a bitmask of the following (defined in **/usr/src/linux/include/linux/if.h**), where IFF_ stands for **interface flags**:

Table 25.5: **netdevice flags**

Field	Meaning
IFF_UP	Interface active.
IFF_BROADCAST	Interface allows broadcasting.
IFF_DEBUG	Turn on debugging (verbose).
IFF_LOOPBACK	Is a loopback interface.
IFF_POINTOPOINT	Indicates a point-to-point link, such as **ppp**.
IFF_NOTRAILERS	Avoid use of trailers. For **BSD** compatibility only.
IFF_RUNNING	Interface resources allocated.
IFF_NOARP	Interface can't perform ARP, such as for **ppp**.
IFF_PROMISC	Interface is operating promiscuously (seeing all packets on the network.)
IFF_ALLMULTI	Interface should receive all multicast packets.
IFF_MASTER	Master interface for load equalization (balance).
IFF_SLAVE	Slave interface for load equalization (balance).

IFF_MULTICAST	Interface capable of multicasting.
IFF_VOLATILE	IFF_LOOPBACK | IFF_POINTOPOINT | IFF_BROADCAST | IFF_MASTER | IFF_SLAVE | IFF_RUNNING
IFF_PORTSEL	Device can switch between different media, such as twisted pair and coax.
IFF_AUTOMEDIA	Automatic media select active.
IFF_DYNAMIC	Address of interface can change, such as a dialup device.

The `features` field is a bit mask of the following potential hardware capabilities of the device:

Table 25.6: **netdevice features**

Field	Meaning
NETIF_F_SG	Can use scatter/gather I/O; can transmit a packet split into distinct memory segments.
NETIF_F_IP_CSUM	Can do checksum of IP packets but not others.
NETIF_F_NO_CSUM	No checksums ever required (such as a loopback device).
NETIF_F_HW_CSUM	Can checksum all packets.
NETIF_F_HIGHDMA	Can DMA to high memory; otherwise all DMA is to low memory.
NETIF_F_FRAGLIST	Can cope with scatter/gather I/O – used in loopback.
NETIF_F_HW_VLAN_TX	Has transmit acceleration for 802.1q VLAN packets.
NETIF_F_HW_VLAN_RX	Has receive acceleration for 802.1q VLAN packets.
NETIF_F_VLAN_FILTER	Can receive filtering on the VLAN.
NETIF_F_VLAN_CHALLENGED	Gets confused and should not handle VLAN packets.
NETIF_F_TSO	Can perform offload TCP/IP segmentation.
NETIF_F_LLTX	Can do lock-less transmission.

The final set of fields are **utility** fields and hold status information. The important ones are:

Table 25.7: **netdevice utility fields**

Field	Meaning
unsigned long trans_start; unsigned long last_rx;	The `jiffies` value when transmission began. (`last_rx` is presently unused.)
int watchdog_timeo;	Minimum time (in `jiffies`) before the `tx_timeout()` function should be called.
void *priv;	A pointer the driver can use at will; a good place to store data.
struct dev_mc_list *mc_list; int mc_count;	Used to handle multicast transmission.
spinlock_t xmit_lock; int xmit_lock_owner;	Used to avoid multiple calls to the transmission function. (Not to be called by the driver itself.)

25.2 net_device_ops Structure

The `netdevice` structure dates back to **Linux**'s earliest days, continuously accreting new fields as new types of devices with new features gained **Linux** support, and as new networking facilities were incorporated.

As part of a move to bring the beast under control a new data structure of type `net_device_ops` was added in the 2.6.29 kernel. It contains the function pointers for the various management hooks for network devices. A pointer to this structure is now contained in the `netdevice` structure. The detailed structure is defined in **/usr/src/linux/include/linux/netdevice.h** and including all conditional fields looks like:

```
struct net_device_ops {
  int  (*ndo_init) (struct net_device *dev);
  void (*ndo_uninit) (struct net_device *dev);
  int  (*ndo_open) (struct net_device *dev);
  int  (*ndo_stop) (struct net_device *dev);
  int  (*ndo_start_xmit) (struct sk_buff *skb, struct net_device *dev);
  u16  (*ndo_select_queue) (struct net_device *dev, struct sk_buff *skb);
  void (*ndo_change_rx_flags) (struct net_device *dev, int flags);
  void (*ndo_set_rx_mode) (struct net_device *dev);
  void (*ndo_set_multicast_list) (struct net_device *dev);
  int  (*ndo_set_mac_address) (struct net_device *dev, void *addr);
```

```
    int  (*ndo_validate_addr) (struct net_device *dev);
    int  (*ndo_do_ioctl) (struct net_device *dev, struct ifreq *ifr, int cmd);
    int  (*ndo_set_config) (struct net_device *dev, struct ifmap *map);
    int  (*ndo_change_mtu) (struct net_device *dev, int new_mtu);
    int  (*ndo_neigh_setup) (struct net_device *dev, struct neigh_parms *);
    void (*ndo_tx_timeout) (struct net_device *dev);
    struct net_device_stats* (*ndo_get_stats) (struct net_device *dev);
    void (*ndo_vlan_rx_register) (struct net_device *dev,struct vlan_group *grp);
    void (*ndo_vlan_rx_add_vid) (struct net_device *dev, unsigned short vid);
    void (*ndo_vlan_rx_kill_vid) (struct net_device *dev, unsigned short vid);
    void (*ndo_poll_controller) (struct net_device *dev);
    int (*ndo_fcoe_ddp_setup) (struct net_device *dev, u16 xid,
                               struct scatterlist *sgl, unsigned int sgc);
    int  (*ndo_fcoe_ddp_done) (struct net_device *dev,u16 xid);
};
```

The header file extensively documents each of these functional methods.

You would initialize the structure with something like:

```
#ifdef HAVE_NET_DEVICE_OPS
static struct net_device_ops ndo = {
    .ndo_open = my_open,
    .ndo_stop = my_close,
    .ndo_start_xmit = stub_start_xmit,
};
#endif
```

and then in your setup routine you have to place the structure in the `net_device` structure, as in:

```
struct net_device *dev;
.....
#ifdef HAVE_NET_DEVICE_OPS
    dev->netdev_ops = &ndo;
#else
    dev->open = my_open;
    dev->stop = my_close;
    dev->hard_start_xmit = stub_start_xmit;
#endif
```

where we show code snippets that will work with older and newer kernel versions.

You can do things either way as long as `CONFIG_COMPAT_NET_DEV_OPS` is set in the kernel configuration file; in version 2.6.31 this option will no longer be available as all in-tree drivers will have been migrated to the new structure.

Other changes to the `netdevice` structure are also in the works, such as moving network protocol information out of it.

25.3 sk_buff Structure

Everything you need to know about socket buffers is contained in the header file **/usr/src/linux/include/linux/skbuff.h**. The `skbuff` structure is another complicated structure, describing the **socket buffer**, which is the data structure that holds a packet.

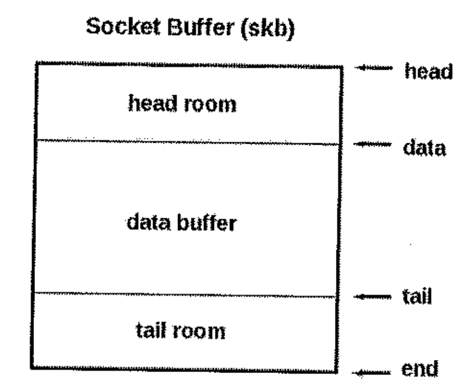


Figure 25.1: **Socket buffer layout**

Leaving out some elements which are used only when **netfilter** is configured, its fields are:

Table 25.8: **Socket buffer fields**

Field	Meaning
`struct sk_buff *next, *prev;`	Next and previous buffers in the linked list.
`struct sk_buff_head *list;`	Linked list of buffers.
`struct sock *sk;`	Socket that owns the packet.
`struct timeval stamp;`	Time the packet arrived.
`struct net_device *dev;`	Device sending or receiving this buffer.
`struct net_device *real_dev;`	Device packet arrived on.

`union { ... } h;`
`union { ... } nh;`
`union { ... } mac;`
	Headers for the transport, network, and link layers. Can be searched for information such as source and destination addresses, etc.

`struct dst_entry *dst;`	Routing information.

`char cb[48];`	A control buffer private variables can be stashed into; some work has to be done to preserve them across layers.
`unsigned int len;`	Length of actual data (`skb->tail - skb->head`)
`unsigned int data_len;`	Length of data buffer(`skb->end - skb->data`)
`unsigned int csum;`	Checksum.
`unsigned char cloned`	It is possible to **clone** the head of the packet.
`unsigned char pkt_type`	Packet class. (`PACKET_HOST, PACKET_BROADCAST, PACKET_MULTICAST, PACKET_OTHERHOST`)
`unsigned char ip_summed;`	Checksum set by the driver on incoming packets.
`__u32 priority;`	Packet queueing priority.
`atomic_t users;`	User reference count.
`unsigned short protocol;`	Packet protocol, from driver.
`unsigned short security;`	Packet security level.
`unsigned int truesize;`	Buffer size.
`unsigned char *head;`	Pointer to head of buffer.
`unsigned char *data;`	Pointer to data head in buffer. Usually slightly greater than **head**.
`unsigned char *tail;`	Pointer to tail of data.
`unsigned char *end;`	Pointer to end of buffer. Maximum address `tail` can reach.
`void (*destructor) (struct sk_buff *);`	Pointer to optional destructor function for packets.

25.4 Socket Buffer Functions

There are a number of functions which are used on socket buffers. They are listed in **/usr/src/linux /include/linux/skbuff.h**, and defined in there or in **/usr/src/linux/net/core/skbuff.c**. The most important ones are:

Table 25.9: **Socket buffer functions**

Type	Function	Use
`struct sk_buff *`	`alloc_skb (unsigned int length);`	Allocate a new socket buffer, give it a reference count of one, and initialize the `data, tail, head` pointers.
`struct sk_buff *`	`dev_alloc_skb (unsigned int length);`	Same as `alloc_skb()` plus the memory is allocated with `GFP_ATOMIC`, so failure results if resources are not immediately available, and some space is reserved between the `head` and `tail` fields of the packet, for optimization use by kernel networking layers.
`void`	`kfree_skb (struct sk_buff *skb);`	Drop the buffer reference count and release it if the usage count is now zero. This form is used by the kernel and is not meant to be used from drivers.

`void`	`dev_kfree_skb (struct sk_buff *skb);`
	`dev_kfree_skb_irq (struct sk_buff *skb);`
	`dev_kfree_skb_any (struct sk_buff *skb);`
		For use in drivers. The three forms are non-interrupt context, interrupt context, or any context.

`unsigned char *`	`skb_put (struct sk_buff *skb, unsigned int len);`	Add `len` bytes of data to the **end** of the buffer, returning a pointer to the first byte of the extra data.
`unsigned char *`	`skb_push (struct sk_buff *skb, unsigned int len);`	Add `len` bytes of data to the **beginning** of the buffer, returning a pointer to the first byte of the extra data.
`unsigned char *`	`skb_pull (struct sk_buff *skb, unsigned int len);`	Remove data from the buffer start, returning a pointer to the new start of data. The space released will go into **headroom**.

`int`	`skb_headroom (const struct sk_buff *skb);`	Returns the number of bytes of free space at the **start** of the buffer.
`int`	`skb_tailroom (const struct sk_buff *skb);`	Returns the number of bytes of free space at the **end** of the buffer.
`void`	`skb_reserve (struct sk_buff *skb, unsigned int len);`	Increase the headroom of an empty buffer by reducing the `tailroom`; buffer must be empty. For instance since Ethernet headers are 14 bytes, reserving 2 bytes would align the following IP header on a 16 byte boundary.
`void`	`skb_trim (struct sk_buff *skb, unsigned int len);`	Remove bytes from the end of a buffer to make it `len` bytes long; if the buffer is already less than this, nothing happens.
`struct sk_buff *`	`skb_get (struct sk_buff *skb);`	Increment reference counter for the buffer and return a pointer to it.
`int`	`skb_shared (struct sk_buff *skb);`	True if more than one person refers to the buffer.
`void`	`skb_orphan (struct sk_buff *skb);`	If the buffer is currently owned, call the owner's destructor function and make the buffer unowned; it will still exist but is not associated with the former owner.
`struct sk_buff *`	`skb_clone (struct sk_buff *skb, gfp_t gfp_mask);`	Duplicate a socket buffer with the clone not owned by any socket. Both copies share the same packet data, but not the structure.
`struct sk_buff *`	`skb_copy (const struct sk_buff *skb, gfp_t gfp_mask);`	Copy a socket buffer including its data, so it can be modified.

There are many other functions for poking into these structures, and for dealing with queues of socket buffers. See ***Corbet, Rubini and Kroah-Hartman*** and the header files for more information.

25.5 Labs

Lab 1: Examining Network Devices

All network devices are linked together in a list. You can get a pointer to the head of the list and then walk through it using:

```
struct net_device *first_net_device (struct net *net);
struct net_device *next_net_device(struct net_device *dev);
```

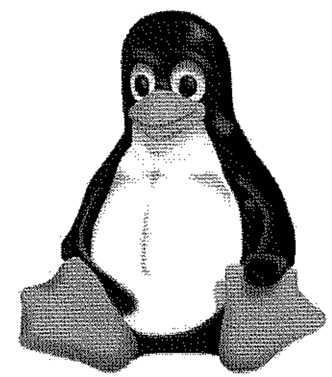
Write a module that works its way down the list and prints out information about each driver.

This should include the name, any associated irq, and various other parameters you may find interesting.

Try doing this with your previous simple network module loaded.

Chapter 26

Network Drivers III: Transmission and Reception



We'll study transmission and reception functions for network device drivers, and how to get statistics on a network driver.

26.1 Transmitting Data and Timeouts

Here's a simple example of a transmission function:

```
int my_hard_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
    int len;
```

```
    char *data;
    struct my_data *priv = netdev_priv (dev);

    len = skb->len ;
    data = skb->data;
    dev->trans_start = jiffies;

    /* so we can free it in interrupt routine */
    priv->skb = skb;

    mynet_hw_tx(data, len, dev);

    return 0 ;
}
....
dev->hard_start_xmit = my_hard_start_xmit;
```

The driver first has to put the data into one or more socket buffers, which are then passed off to a hardware-specific function, `mynet_hw_tx()` (which you will have to write) which is responsible for actually getting the data onto the device. When it is finished transmitting an interrupt will be issued and the socket buffers can be freed in the interrupt handler

A timestamp (the present `jiffies` value) is placed in the `dev->trans_start` value, and the private data section of the device structure (`dev->priv`), which has previously been declared to be some kind of data structure including a `skb` field, is also used.

The above simple function can't be the whole story. In particular one has to deal with **transmission timeouts**, or any condition in which the device fails to respond.

When we initialized the `net_device` structure, we filled in a field for `dev->watchdog_timeo`. Whenever a time value greater than the value for this field has elapsed since `dev->trans_start`, and the socket buffer hasn't been freed by the hardware transmission function, the kernel networking code will cause the `tx_timeout()` function associated with the driver to be called. This function has to do whatever is necessary to ensure proper completion of any in-progress transmissions, and do whatever is necessary to clear up the problem.

One problem that can develop is that the outgoing device generally has only a limited amount of memory to store outgoing packets; when that memory is filled, the driver will have to call `netif_stop_queue()` to notify the kernel it can't accept any more outgoing data, and then call `netif_wake_queue()` when it is again ready to accept. Whether the number of possible outgoing packets is as few as one, or is many, drivers have to be prepared to deal with this eventuality.

Occasionally one might need to disable packet transmission from somewhere else than in the `hard_start_xmit()` callback function. An example would be in response to a re-configuration request. Rather than using `netif_stop_queue()`, one should employ the function:

```
void netif_tx_disable (struct net_device *dev);
```

which avoids race conditions by making sure the `hard_start_xmit()` function is not already running on another CPU when it returns. The queue is still woken up as usual with `netif_wake_queue()`.

26.2 Receiving Data

The interrupt handler will be invoked upon receipt of an interrupt from the network device, which will determine if it is being called because data has arrived, or because data has been sent. (Normally it will do this by checking some registers.)

Either the handler or the transmission routine it calls must allocate (or reuse) any necessary socket buffers. Using `dev_alloc_skb()` for this allocation can be done at interrupt time since it uses `GFP_ATOMIC`.

When the packet has been successfully obtained, the function `netif_rx()` gets called to pass the buffer up to the higher network layers in the kernel.

Here's an very simple example (without error checking) of a routine which would get called out of the interrupt handler, which takes as arguments a pointer to the network device, and one to a buffer of data of known length taken off the device:

```
void mynet_rx(struct net_device *dev, int len, unsigned char *buf)
{
    struct sk_buff *skb;
    skb = dev_alloc_skb(len+2);
    skb_reserve(skb, 2);
    memcpy(skb_put(skb, len), buf, len);
    skb->dev = dev;
    skb->protocol = eth_type_trans(skb, dev);
    skb->ip_summed = CHECKSUM_UNNECESSARY;
    dev->stats.rx_packets++;
    netif_rx(skb);
    return;
}
```

Since the Ethernet header is 14 bytes long, the `skb_reserve()` function is called to pad out so the Internet header can be put on a word boundary. The job of the driver is done when the socket buffer is passed upward and onward by `netif_rx()`.

26.3 Statistics

Statistics are stored in a structure, accessed from the of type `net_device_stats`. They are accessed through the `dev->net_device_stats` function field, which should return a pointer to the data structure. For example:

```
....
dev->get_stats = mynet_stats;
...
static struct net_device_stats *mynet_stats (struct net_device *dev)
{
        return &dev->stats;
}
```

The `net_device_stats` data structure looks like:

```
struct net_device_stats
{
  unsigned long   rx_packets;         /* total packets received       */
  unsigned long   tx_packets;         /* total packets transmitted    */
  unsigned long   rx_bytes;           /* total bytes received         */
  unsigned long   tx_bytes;           /* total bytes transmitted      */
  unsigned long   rx_errors;          /* bad packets received         */
  unsigned long   tx_errors;          /* packet transmit problems     */
  unsigned long   rx_dropped;         /* no space in linux buffers    */
  unsigned long   tx_dropped;         /* no space available in linux  */
  unsigned long   multicast;          /* multicast packets received   */
  unsigned long   collisions;

  /* detailed rx_errors: */
  unsigned long   rx_length_errors;
  unsigned long   rx_over_errors;     /* receiver ring buff overflow  */
  unsigned long   rx_crc_errors;      /* recved pkt with crc error    */
  unsigned long   rx_frame_errors;    /* recv'd frame alignment error */
  unsigned long   rx_fifo_errors;     /* recv'r fifo overrun          */
  unsigned long   rx_missed_errors    /* receiver missed packet       */

  /* detailed tx_errors */
  unsigned long   tx_aborted_errors;
  unsigned long   tx_carrier_errors;
  unsigned long   tx_fifo_errors;
  unsigned long   tx_heartbeat_errors;
  unsigned long   tx_window_errors;

  /* for cslip etc */
  unsigned long   rx_compressed;
  unsigned long   tx_compressed;
};
```

The command `ifconfig eth{n}` will generate a report of these statistics, which you can update in your driver. These can be garnered by looking at the **/proc/net/dev** entry.

26.4 Labs

Lab 1: Building a Transmitting Network Driver

Extend your stub network device driver to include a transmission function, which means supplying a method for `dev->hard_start_xmit()`.

While you are at it, you may want to add other entry points to see how you may exercise them.

Once again, you should be able to exercise it with:

```
insmod lab1_network.ko
ifconfig mynet0 up 192.168.3.197
ping -I mynet0 localhost
   or
ping -bI mynet0 192.168.3
```


Make sure your chosen address is not being used by anything else.

Lab 2: Adding Reception

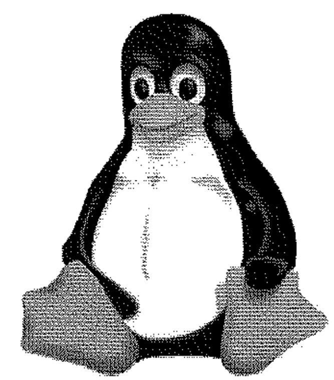
Extend your transmitting device driver to include a reception function.

You can do a **loopback** method in which any packet sent out is received.

Be careful not to create memory leaks!

Chapter 27

Network Drivers IV: Selected Topics



We'll consider the use of **multicasting**, changes in the carrier state of the device, and the use of **ioctl()** commands. We'll also consider the questions of interrupt mitigation, **TSO** and **TOE**, and **MII** and **ethtool** support.

27.1 Multicasting

A **multicast** network packet is sent to more than one (but not all) network destinations. For this purpose a unique hardware address is assigned to a group of hosts; any packet sent to that address will be received by all members of the group.

For Ethernet this requires that the least significant bit in the first byte of the destination address is set; at the same time the first bit in the first byte of the device hardware address is cleared.

Multicast packets look no different than any other packet, and thus a device driver need do nothing special to transmit them. It is up to the kernel to route them to the right hardware addresses.

However, reception of multicast packets is more complex and can require more or less work from a network device, depending on its sophistication.

Some devices have no special multicast capability. They receive packets that are either sent directly to their hardware address, or broadcast to all addresses. They receive multicast traffic only by receiving all packets; this can overwhelm the system. Such a device will not have the `IFF_MULTICAST` flag set in its `net_device` structure.

A second class of device can distinguish multicast packets from ordinary ones; they receive every one and let software decide whether or not is intended for them and should be taken in. This has a lower overhead than the first class.

A third class of device performs multicast packet detection in its hardware. This kind of device accepts a list of multicast addresses to be interested in and ignores all others. This is the most efficient class of device because it doesn't bother accepting packets and then dropping them. Whenever the list of valid multicast addresses is modified, the kernel updates the list the device is aware of.

The method called whenever the list of multicast machine addresses the device is associated with changes is:

```
void (*dev->set_multicast_list)(struct net_device *dev);
```

(where `dev` is the device's `net_device` structure). The function is invoked whenever `dev->flags` changes. If the driver can't implement this method it should just supply `NULL`.

The data structure giving the linked list of all multicast addresses the device deals with is:

```
struct dev_mc_list
{
   struct dev_mc_list   *next;                /* next address in list */
   __u8                 dmi_addr[MAX_ADDR_LEN]; /* hardware address */
   unsigned char        dmi_addrlen;          /* address length */
   int                  dmi_users;            /* # of users */
   int                  dmi_gusers;           /* # of groups */
};

struct dev_mc_list *dev->mc_list;
```

There are also a number of flags (in `dev->flags`) which affect behaviour:

Table 27.1: **Multicasting flags**

Flag	Meaning
`IFF_MULTICAST`	If not set, the device won't handle multicast packets. However, the `set_multicast_list()` method will still be called whenever the flags change.
`IFF_ALLMULTI`	Tells the driver to retrieve **all** multicast packets.
`IFF_PROMISC`	Puts the interface in **promiscuous** mode. All packets are received regardless of what is in `dev->mc_list`.

Corbet, Rubini and Kroah-Hartman give an example of an implementation of the `set_multicast_list()` method.

27.2 Changes in Link State

Network connections can go up and down due to external events, such as plugging a cable in and out. Almost all network devices have a capability of sensing a **carrier** state; when present it means the hardware is available.

Linux provides the following functions to notify the other networking layers when the state goes up or down or to inquire about it:

```
void netif_carrier_on  (struct net_device *dev);
void netif_carrier_off (struct net_device *dev);
int  netif_carrier_ok  (struct net_device *dev);
```

The on and off functions should be called whenever the driver detects a change of state. They may also be called to bracket a major configuration change, or reset.

The final function just checks the `net_device` structure to sense the state.

27.3 ioctls

When an `ioctl()` command is passed to a socket descriptor, the command is first compared to those defined in **/usr/src/linux/include/linux/sockios.h**.

If the command is a socket configuration command, such as `SIOCSIFADDR` it is directly acted upon by high levels of the networking code.

The command may also be one of the protocol-specific functions defined in the header file. In either case, the third argument to `ioctl()` is cast as a pointer to a structure of type `struct ifreq`, defined in **/usr/src/linux/include/linux/if.h**:

If the kernel doesn't recognize the command, it is passed to the do_ioctl() method defined in the driver.

```
int (*do_ioctl) (struct net_device *dev, struct ifreq *ifr, int cmd);
```

For this purpose 16 commands are seen as private to the device (SIOCDEVPRIVATE through SIOCDEVPRIVATE+15.)

Note that the ifr field actually points to a kernel-space copy of the user-passed data structure. Thus the driver can freely use this structure without resort to functions like copy_to_user().

27.4 NAPI and Interrupt Mitigation

The straightforward way to write a network device driver is to have the arrival of each packet accompanied by an interrupt. The interrupt handler does the necessary work (including queueing up work for deferred processing, perhaps through a tasklet) and then is ready for more data.

For high-bandwidth devices such an approach starts to cause problems; even if the full bandwidth can be maintained, the amount of CPU time expended to keep up with it can start to overwhelm the system.

The 2.6 kernel contains an alternative method, or interface, based on **polling** the device. This interface is called **NAPI** (New **API**) for lack of a better name.

A device capable of using **NAPI** must be able to store some number of packets (either on board on an in-memory **DMA** ring buffer). It also must be capable of disabling interrupts for packet reception while continuing to issue interrupts for successful transmissions (and possibly other events.)

Given these capabilities, a **NAPI**-based driver turns off reception interrupts, and periodically polls and consumes accumulated events. When traffic slows down, normal interrupts are turned back on and the driver functions in the old-fashioned way.

One should repeat that **NAPI**- based drivers exist for only a few high bandwidth devices, and in general do not directly improve throughput. However they very significantly cut down on CPU load.

27.5 NAPI Details

Assuming one has the necessary hardware capabilities for the device (the ability to turn off interrupts for incoming packets and to store a sufficient amount of data packets), the first step in writing a network driver that includes interrupt mitigation is to add two new fields to the net_device data structure:

```
dev->poll = my_poll;
dev->weight = my_weight;
```

The poll() method will handle the data that has accumulated while incoming data interrupts are turned off. The weight parameter indicates how much traffic should be accepted through the interface; for 10 MB interfaces it should be set to 16; faster interfaces should use 64. The weight field should not be set to a number greater than the number of packets the interface can store.

One must also rewrite the interrupt handling routine so that when an incoming packet is received (with incoming interrupts enabled obviously) it should turn off further reception interrupts, and hand the packet off to the function netif_rx_schedule (struct net_device *) which will induce the poll() method to be called eventually.

The poll() method looks like:

```
int (*poll)(struct net_device *dev, int *budget);
```

where the budget argument is the maximum number of packets which the function can process. As packets are processed, they are fed to netif_receive_skb(), not to netif_rx() as in normal reception functions.

If the method is able to process all available packets it turns reception interrupts back on, calls the function netif_rx_complete() to turn off polling, and returns a value of 0. A return value of 1 indicates more packets need to be processed.

27.6 TSO and TOE

The purpose of **TSO** (TCP Segmentation Offload) is to allow a buffer much larger than the usual **MTU** (maximum transfer unit), which is usually only 1500 bytes, to be passed to the network device.

The breaking down (segmentation) of the large buffer into smaller mtu-limited segments than can pass through routers, switches, etc, is normally done by the CPU before data is passed to the device.

However, with **TSO**, a 64 KB buffer is broken into 44 mtu-sized segments on the device itself. Each fragment, or packet, has attached to it the **TCP** and **IP** protocol headers, using a template.

The net result is a potentially large reduction in the load on the CPU rather than an increase of bandwidth, as is the case with many advanced techniques. Generally **TSO** will be important only for high-bandwidth, such as 1 GB or greater network devices.

The purpose of **TOE** (TCP/IP Offload Engine) technology is to move TCP/IP processing to an integrated circuit on board the network card. As with **TSO**, the idea is to free up the CPU to do other work.

Unlike **TSO**, the **TOE** mechanism affects both inbound and outbound traffic. Because it is a connection-oriented protocol, there is a lot of complexity involved.

In **Linux**, however, kernel developers have rejected inclusion of **TOE** while they have heartily embraced **TSO**. One reason is that performance levels are not enhanced enough to justify the complexity.

Furthermore, since the TCP/IP stack is implemented on the card, often in a black box with closed source, it is difficult to keep security up to date and have behaviour match expectations.

Resource limitations (such as the number of simultaneous connections that can be handled) are more limited than they are for **Linux** in general; this can be used to facilitate denial of service attacks.

For a detailed explanation of why **TOE** will never be accepted in the main **Linux** kernel, see **http://linux-net.osdl.org/index.php/TOE**.

27.7 MII and ethtool

Many network devices comply with the **MII** (Media Independent Interface) standard, which describes the interface between network controllers and **Ethernet** transceivers.

The kernel supports the generic **MII** interface with:

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

struct mii_if_info {
   int phy_id;
   int advertising;
   int phy_id_mask;
   int reg_num_mask;

   unsigned int full_duplex : 1;   /* is full duplex? */
   unsigned int force_media : 1;   /* is autoneg. disabled? */

   struct net_device *dev;
   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);
};
```

The key methods embedded in this structure are `mdio_read()` and `mdio_write()`, which take care of communications with the interface. There exist other functions for obtaining information about and changing the device state, which are also designed to collaborate with the **ethtool** utility.

ethtool affords system administrators with a handy set of utilities for controlling interface attributes, such as speed, media type, duplex operation, checksumming, etc. The driver must have direct support for **ethtool** to take full advantage of its features.

The relevant code is found in **/usr/src/linux/include/linux/ethtool.h**, and includes the `ethtool_ops` structure, which contains a list of methods that can be implemented:

```
struct ethtool_ops {
   int  (*get_settings)(struct net_device *, struct ethtool_cmd *);
   int  (*set_settings)(struct net_device *, struct ethtool_cmd *);
   void (*get_drvinfo)(struct net_device *, struct ethtool_drvinfo *);
   int  (*get_regs_len)(struct net_device *);
   void (*get_regs)(struct net_device *, struct ethtool_regs *, void *);
   void (*get_wol)(struct net_device *, struct ethtool_wolinfo *);
   int  (*set_wol)(struct net_device *, struct ethtool_wolinfo *);
   u32  (*get_msglevel)(struct net_device *);
   void (*set_msglevel)(struct net_device *, u32);
   int  (*nway_reset)(struct net_device *);
   u32  (*get_link)(struct net_device *);
   int  (*get_eeprom_len)(struct net_device *);
   int  (*get_eeprom)(struct net_device *, struct ethtool_eeprom *, u8 *);
   int  (*set_eeprom)(struct net_device *, struct ethtool_eeprom *, u8 *);
   int  (*get_coalesce)(struct net_device *, struct ethtool_coalesce *);
   int  (*set_coalesce)(struct net_device *, struct ethtool_coalesce *);
   void (*get_ringparam)(struct net_device * struct ethtool_ringparam *);
   int  (*set_ringparam)(struct net_device * struct ethtool_ringparam *);
```

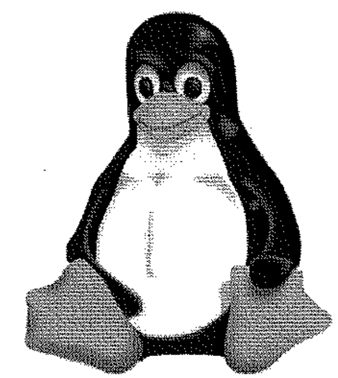


```
   void (*get_pauseparam)(struct net_device *, struct ethtool_pauseparam*);
   int  (*set_pauseparam)(struct net_device *, struct ethtool_pauseparam*);
   u32  (*get_rx_csum)(struct net_device *);
   int  (*set_rx_csum)(struct net_device *, u32);
   u32  (*get_tx_csum)(struct net_device *);
   int  (*set_tx_csum)(struct net_device *, u32);
   u32  (*get_sg)(struct net_device *);
   int  (*set_sg)(struct net_device *, u32);
   u32  (*get_tso)(struct net_device *);
   int  (*set_tso)(struct net_device *, u32);
   int  (*self_test_count)(struct net_device *);
   void (*self_test)(struct net_device *, struct ethtool_test *, u64 *);
   void (*get_strings)(struct net_device *, u32 stringset, u8 *);
   int  (*phys_id)(struct net_device *, u32);
   int  (*get_stats_count)(struct net_device *);
   void (*get_ethtool_stats)(struct net_device *, struct ethtool_stats *, u64 *);
   int  (*begin)(struct net_device *);
   void (*complete)(struct net_device *);
};
```

To enable **ethtool** for your device you have to set a pointer to this structure in the `netdevice` structure, using the `SET_ETHTOOL_OPS` macro. If **MII** support is also enabled, the functions `mii_ethtool_gset()` and `mii_ethtool_sset()` can be used to implement the `get_settings()` and `set_settings()` methods.

Chapter 28

USB Drivers



We'll discuss **USB** devices, what they are, the standard that describes them, the topology of the connection of hubs, peripherals and host controllers, and the various descriptors inolved. We'll consider the different kinds of classes and data transfers possible. Then we'll see how **USB** has been implemented under **Linux**. We review registration/deregistration of **USB** devices. We'll describe the entry points to the driver and some of the main functions and data structures in the **USB API**. Finally, we'll do a code walkthrough on a simple **USB** driver.

28.1 What is USB?

USB stands for Universal Serial Bus. It permits easy connection of multiple peripheral devices to one port, and automatic, hotplug, configuration of devices attached (and detached) while the computer is running. Virtually any type of peripheral (with **USB** capability) can be connected to a **USB** port; i.e., scanners, modems, network cards, mice, keyboards, printers, mass storage devices, etc.

Version 1.0 of the **USB** specification was released in January 1996 by an alliance of Compaq, Intel, Microsoft and NEC. Version 1.1 was released in September 1998, and version 2.0 was released in 1999.

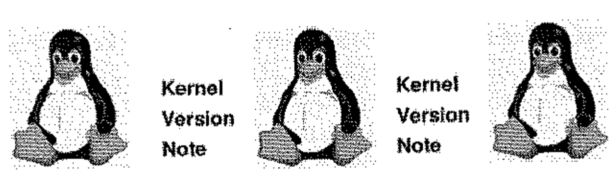
One thing to be careful about is when considering the **USB** 2.0 standard, is that when the phrase ***full speed*** or ***low speed*** is used, it stands in for **USB** 1.1. The newer standard is described as ***high speed***.

Up to 127 devices can be connected simultaneously. The **USB** cable contains four wires; power, ground and two signal wires. In the original standard, the ideal total bandwidth was limited to 12 Mbit/s, but overheads limited this to something like 8.5 Mbit/s and realistic performance was probably as low as 2 Mbit/s. **USB** 2.0 brought a theoretical speed limit of 480 Mbit/s.

Devices may be either low or high speed, or operate in either mode according to function. A high speed device hooked up to a **USB** 1.1 controller or hub will be limited to lower capabilities.

Power can be delivered either through the **USB** cable or through a peripheral's own power supply. A total of up to 500 mA can be supplied through each controller. When a device is plugged in it can initially grab up to 100 mA and then request more if limits are not exceeded.

The kernel contains a lot of **USB**-related documentation in the **/usr/src/linux/Documentation /usb** directory.



- Support for the new **USB** 3.0 standard (also known as **XHCI**) has been included in kernel version 2.6.31, making **Linux** the first operating system to incorporate it.

28.2 USB Topology

USB ports are incorporated in all recent motherboards. In most cases there are at least 2 ports. The ports can be connected either directly to devices or to hubs, which themselves can be connected to more hubs or devices. There is a ***virtual root hub*** simulated by the host controller. The total number of ports plus hubs is 127.

Technically, the physical structure of **USB** is not that of a bus; it is a tree with **upstream** and **downstream** nodes. Each device can have only one upstream connection (with a type ***A*** connector),

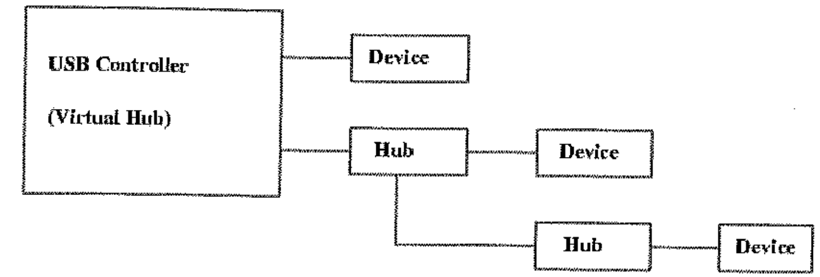


Figure 28.1: **USB topology**

but a hub node can have more than one downstream connection (with a type ***B*** connector.)

For **USB** 1.x, there are two types of host controllers:

- **OHCI** (Open Host Controller Interface) from Compaq.
- **UHCI** (Universal Host Controller Interface) from Intel.

UHCI is simpler and thus requires a somewhat more complex device driver. Peripherals should work equally well with either controller.

For **USB** 2.0, the standard is **EHCI** (Enhanced Host Controller Interface.)

Upon being hooked up to the bus, a peripheral identifies itself as belonging to one of several classes. When a particular driver is loaded it will claim the device and handle all communication with it.

28.3 Descriptors

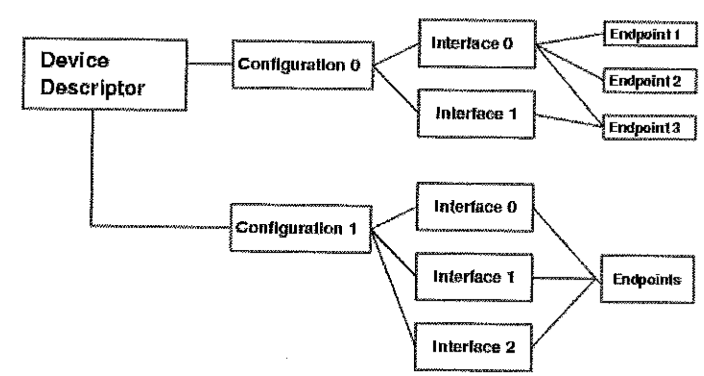


Figure 28.2: **USB descriptors**

Each **USB** device has a unique **device descriptor**, assigned to it when the peripheral is connected to the bus. In addition it gets a device number assigned (an integer ranging from 1 to 127). The descriptor has all pertinent information applying to the device and all of its possible configurations.

A device has one or more **configuration descriptors**. This has specific information about how the device may be used.

Each configuration points to one or more **interface descriptors**. Each interface might point to various alternate settings about how the device might be used. For instance, a video camera could have three alternate settings, which require different bandwidths: camera activated, microphone activated, and camera and microphone activated.

Each interface points to one or more **endpoint descriptors**, which give the data source or sink of the device. (All control transfers use an end point of zero.)

28.4 USB Device Classes

If a device plugged into the **USB** hub belongs to a well-known **device class**, it is expected to conform to certain standards with respect to device and interface descriptors. Thus the same device driver can be used for any device that claims to be a member of that class. The following classes are defined in the standard:

Table 28.2: **USB device classes**

Base Class	Descriptor Usage	Description	Examples
00h	Device	Unspecified	Use class information in the interface descriptors
01h	Interface	Audio	speakers, microphones. sound cards
02h	Both	Communications and CDC Control	network adapters, modems, serial port adapters
03h	Interface	HID (Human Interface Device)	mice, joysticks, keyboards
05h	Interface	Physical	force feedback joystick
06h	Interface	Image	digital cameras
07h	Interface	Printer	printers
08h	Interface	Mass Storage	flash drives, MP3 players, memory card readers
09h	Device	Hub	full and high speed hubs
0Ah	Interface	CDC-Data	used together with CDC Control
0Bh	Interface	Smart Card	smart card readers
0Dh	Interface	Content Security	security
0Eh	Interface	Video	webcams
0Fh	Interface	Personal Healthcare	healthcare devices
DCh	Both	Diagnostic Device	USB compliance testing devices
E0h	Interface	wireless controller	bluetooth and wi-fi adapters
EFh	Both	Miscellaneous	ActiveSync devices
FEh	Interface	Application Specific	irda bridge
FFh	Both	Vendor Specific	devices needing vendor specific drivers

Other devices require a fully customized device driver be written.

28.5 Data Transfer

There are the following types of data transfer to and from **USB** devices:

Control transfers are short commands that configure and obtain the state of devices. While there may also be device specific commands, most or all devices will support the following standard set defined in **/usr/src/linux/include/linux/usb_ch9.h**:

```
2.6.31:  79 #define USB_REQ_GET_STATUS          0x00
2.6.31:  80 #define USB_REQ_CLEAR_FEATURE       0x01
2.6.31:  81 #define USB_REQ_SET_FEATURE         0x03
2.6.31:  82 #define USB_REQ_SET_ADDRESS         0x05
2.6.31:  83 #define USB_REQ_GET_DESCRIPTOR      0x06
2.6.31:  84 #define USB_REQ_SET_DESCRIPTOR      0x07
2.6.31:  85 #define USB_REQ_GET_CONFIGURATION   0x08
2.6.31:  86 #define USB_REQ_SET_CONFIGURATION   0x09
2.6.31:  87 #define USB_REQ_GET_INTERFACE       0x0A
2.6.31:  88 #define USB_REQ_SET_INTERFACE       0x0B
2.6.31:  89 #define USB_REQ_SYNCH_FRAME         0x0C
2.6.31:  90                                /* Wireless USB */
2.6.31:  91 #define USB_REQ_SET_ENCRYPTION      0x0D
2.6.31:  92 #define USB_REQ_GET_ENCRYPTION      0x0E
2.6.31:  93 #define USB_REQ_RPIPE_ABORT         0x0E
2.6.31:  94 #define USB_REQ_SET_HANDSHAKE       0x0F
2.6.31:  95 #define USB_REQ_RPIPE_RESET         0x0F
2.6.31:  96 #define USB_REQ_GET_HANDSHAKE       0x10
2.6.31:  97 #define USB_REQ_SET_CONNECTION      0x11
2.6.31:  98 #define USB_REQ_SET_SECURITY_DATA   0x12
2.6.31:  99 #define USB_REQ_GET_SECURITY_DATA   0x13
2.6.31: 100 #define USB_REQ_SET_WUSB_DATA       0x14
2.6.31: 101 #define USB_REQ_LOOPBACK_DATA_WRITE 0x15
2.6.31: 102 #define USB_REQ_LOOPBACK_DATA_READ  0x16
2.6.31: 103 #define USB_REQ_SET_INTERFACE_DS    0x17
```

Bulk transfers send information using up to the full bandwidth. These are *reliable* (i.e., they are checked) and are used by devices like scanners.

Interrupt transfers also can take up to the full bandwidth, but they are sent in response to periodic polling. If the transfer is interrupted, the host controller will repeat the request after a set interval.

Isochronous transfers take up to the full bandwidth as well, but are not guaranteed to be *reliable*. Multimedia devices, audio, video, use these.

28.6 USB under Linux

There are three layers in the **USB** stack under **Linux**:

- Host Controller Driver (OHCI, UHCI, EHCI).
- **USB** Core.
- Device Drivers.

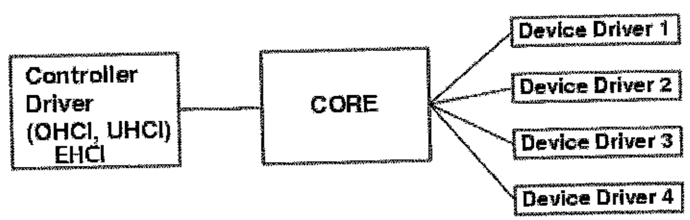


Figure 28.3: **USB: Controller, Core and Device**

The **USB** core has **API**'s for both the controller drivers and the device drivers. It can be thought of as a library of common routines that both the controller and the peripherals can utilize.

Device drivers need not concern themselves with the parts of **API** that interact with the host controller. The driver interacts with the **Linux** kernel by going through the **USB** core.

28.7 Registering USB Devices

USB devices are registered and unregistered with the following functions:

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

int  usb_register   (struct usb_driver *drv);
void usb_deregister (struct usb_driver *drv);
```

`usb_register()` returns 0 for success, a negative number for failure.

Before the device is registered the all-important `usb_driver` structure must be fully initialized. It can be found in **/usr/src/linux/include/linux/usb.h** and looks like:


```
struct usb_driver {
    const char *name;
    int (*probe) (struct usb_interface *intf, const struct usb_device_id *id);
    void (*disconnect) (struct usb_interface *intf);
    int (*ioctl) (struct usb_interface *intf, unsigned int code, void *buf);
    int (*suspend) (struct usb_interface *intf, pm_message_t message);
    int (*resume) (struct usb_interface *intf);
    int (*reset_resume) (struct ubs_interface *intf);
    const struct usb_device_id *id_table;
    struct usb_dynids dynids;
    struct usbdrv_wrap drvwrap;
    unsigned int no_dynamic_id:1;
    unsigned int supports_autsuspend:1;
    unsigned int soft_unbind:1;
};
```

`name` is the name of the driver module. It must be unique among all **USB** drivers, and will show up in **/sys/bus/usb/drivers** when the driver is loaded.

`probe()` points to the function used to check for the device, and is called when new devices are added to the **USB** bus. If the driver feels it can claim the device (based on the information in the `usb_device_id` structure passed to it), the routine should initialize the device and return zero. If the driver does not claim the device, it should return a negative error value.

`disconnect()` points to the function called when devices are removed from the **USB** bus, while `suspend()` and `resume()` point to the functions used for power management.

`ioctl()` will get called when a user-space program issues an `ioctl()` command on the **usbfs** filesystem entry associated with a device attached to this driver. In practice, only the **USB core** uses it for things like hubs and controllers.

Only the main fields have to be set; others are more optional. Thus one might have:

```
static struct usb_driver my_usb_driver = {
    .name       = "my_usb_device",
    .id_table   = my_usb_id_table,
    .probe      = my_usb_probe,
    .disconnect = my_usb_disconnect,
}
```

`id_table` points to an identifying structure of type `usb_device_id`, defined in **/usr/src/linux /include/linux/mod_devicetable.h**.

```
struct usb_device_id {
      /* which fields to match against? */
       __u16           match_flags;

      /* Used for product specific matches; range is inclusive */
       __u16           idVendor;
       __u16           idProduct;
       __u16           bcdDevice_lo;
       __u16           bcdDevice_hi;

      /* Used for device class matches */
```

```
    __u8            bDeviceClass;
    __u8            bDeviceSubClass;
    __u8            bDeviceProtocol;

    /* Used for interface class matches */
    __u8            bInterfaceClass;
    __u8            bInterfaceSubClass;
    __u8            bInterfaceProtocol;

    /* not matched against */
    kernel_ulong_t  driver_info;
};
```

match_flags sets which of the other fields in the structure the device should be matched against. Usually this is not set directly; it is initialized by the USB_DEVICE macros we'll discuss.

idVendor is the unique **USB** vendor ID for the device, assigned by the **USB** controlling body. idProduct is the product ID and is set by the vendor.

bcDevice_lo, bcDevice_hi are the low and high ends of the vendor-determined product number

bDeviceClass, bDeviceSubClass, bDeviceProtocol are assigned by the **USB** controlling body and describe the whole device, including all interfaces.

bInterfaceClass, bInterfaceSubClass, bInterfaceProtocol describe the individual interface.

driver_info can be used by the driver to distinguish different devices from each other when probe() is called.

The usb_device_id table is usually initialized using the following macros:

```
USB_DEVICE(vendor, product)
USB_DEVICE_VER(vendor, product, lo, hi)
USB_DEVICE_INFO(class, subclass, protocol)
USB_INTERFACE_INFO(class, subclass, protocol)
```

in a straightforward way. Thus for a simple driver controlling only one device from one vendor you might have:

```
static struct usb_device_id my_usb_id_table = {
   { USB_DEVICE(USB_MY_VENDOR_ID, MY_VENDOR_PRODUCT_ID) },
   { }  /* Null terminator (required) */
};
MODULE_DEVICE_TABLE (usb, my_usb_id_table);
```

The MODULE_DEVICE_TABLE macro is necessary for user-space hotplug utilities to do their work.

You may have noticed that the callback functions don't refer directly to the usb_driver structure, but instead point to a structure of type usb_interface. One can go back and forth between the two structures with:

```
struct usb_device *interface_to_usbdev (struct usb_interface *intf);
void *usb_get_intfdata (struct usb_interface *intf);
void *usb_set_intfdata (struct usb_interface *intf);
```


The first function retrieves a pointer to the underlying usb_device.

The other two functions get and set a pointer to the data element within the struct device_driver, pointed to in the struct usb_driver pointed to by the struct usb_interface. With this private data pointer nested so deep within the structures, these functions are quite useful.

The no_dynamic_id field lets a driver disable addition of dynamic device IDs.

28.8 Example of a USB Driver

In order to see how it all fits together, let's take a look at **/usr/src/linux/drivers/usb/misc /rio500.c**, a relatively simple driver for a type of **MP3** device that attaches to the **USB** port.

We will only look at the part of the driver that handles initializing, registering, and probing and disconnecting. The actual data transfer code will of course be quite hardware dependent. It is composed mostly of entry point functions pointed to in the file_operations jump table.

Note that one has reference to a file_operations structure as in a character device:

```
2.6.31: 432 static struct
2.6.31: 433 file_operations usb_rio_fops = {
2.6.31: 434     .owner =          THIS_MODULE,
2.6.31: 435     .read =           read_rio,
2.6.31: 436     .write =          write_rio,
2.6.31: 437     .unlocked_ioctl = ioctl_rio,
2.6.31: 438     .open =           open_rio,
2.6.31: 439     .release =        close_rio,
2.6.31: 440 };
2.6.31: 441
2.6.31: 442 static struct usb_class_driver usb_rio_class = {
2.6.31: 443     .name =           "rio500%d",
2.6.31: 444     .fops =           &usb_rio_fops,
2.6.31: 445     .minor_base =     RIO_MINOR,
2.6.31: 446 };
```

The file_operations structure is pointed to by an entry in the struct usb_class_driver, which will be associated with the device in the usb_register_dev() function call, which is made from the probe() callback function.

There is also a structure of type usb_driver which points to the callback functions:

```
2.6.31: 514 static struct usb_device_id rio_table [] = {
2.6.31: 515     { USB_DEVICE(0x0841, 1) },                /* Rio 500 */
2.6.31: 516     { }                                       /* Terminating entry */
2.6.31: 517 };
2.6.31: 518
2.6.31: 519 MODULE_DEVICE_TABLE (usb, rio_table);
2.6.31: 520
2.6.31: 521 static struct usb_driver rio_driver = {
2.6.31: 522     .name =          "rio500",
2.6.31: 523     .probe =         probe_rio,
2.6.31: 524     .disconnect =    disconnect_rio,
```

```
	.id_table =	rio_table,
};
```

Note that the init function simply registers the usb_driver structure:

```
static int __init usb_rio_init(void)
{
	int retval;
	retval = usb_register(&rio_driver);
	if (retval)
		goto out;

	printk(KERN_INFO KBUILD_MODNAME ": " DRIVER_VERSION ":"
		DRIVER_DESC "\n");

out:
	return retval;
}
```

Likewise, the cleanup, or exit, function simply unregisters:

```
static void __exit usb_rio_cleanup(void)
{
	struct rio_usb_data *rio = &rio_instance;

	rio->present = 0;
	usb_deregister(&rio_driver);


}
```

The real work is done by the probe() and disconnect() functions, as far as setting things up and freeing resources. Note these entry points are called by the **USB** core, not user-space programs.

The probe() and disconnect() functions are:

```
static int probe_rio(struct usb_interface *intf,
			const struct usb_device_id *id)
{
	struct usb_device *dev = interface_to_usbdev(intf);
	struct rio_usb_data *rio = &rio_instance;
	int retval;

	dev_info(&intf->dev, "USB Rio found at address %d\n", dev->devnum);

	retval = usb_register_dev(intf, &usb_rio_class);
	if (retval) {
		err("Not able to get a minor for this device.");
		return -ENOMEM;
	}

	rio->rio_dev = dev;
```



```

	if (!(rio->obuf = kmalloc(OBUF_SIZE, GFP_KERNEL))) {
		err("probe_rio: Not enough memory for the output buffer");
		usb_deregister_dev(intf, &usb_rio_class);
		return -ENOMEM;
	}
	dbg("probe_rio: obuf address:%p", rio->obuf);

	if (!(rio->ibuf = kmalloc(IBUF_SIZE, GFP_KERNEL))) {
		err("probe_rio: Not enough memory for the input buffer");
		usb_deregister_dev(intf, &usb_rio_class);
		kfree(rio->obuf);
		return -ENOMEM;
	}
	dbg("probe_rio: ibuf address:%p", rio->ibuf);

	mutex_init(&(rio->lock));

	usb_set_intfdata (intf, rio);
	rio->present = 1;

	return 0;
}
```

28.9 Labs

Lab 1: Installing a USB device.

We are going to write a simple **USB** device driver.

The driver should register itself with the **USB** subsystem upon loading and unregister upon unloading.

The probe() and disconnect() functions should issue printout whenever the device is added or removed from the system.

Your instructor will pass around one or more **USB** devices, such as web cameras, keyboards and mice.

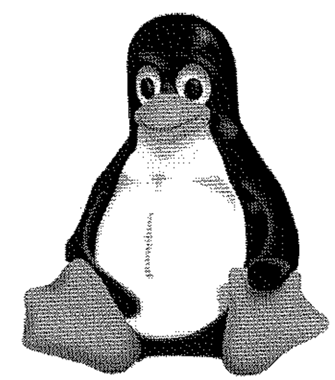
By proper use of the usb_device_id table, you can configure your driver either to sense any device plugged, or only a specific one. You can obtain the vendor and device ID's by noting the output when the **USB** subsystem senses device connection.

You will have to make sure your kernel has the proper **USB** support compiled in, and that no driver for the device is already loaded, as it may interfere with your driver claiming the device.

Hint: You'll probably want to do a make modules_install to get automatic loading to work properly.

Chapter 29

Memory Technology Devices



We are going to consider the different types of **MTD** devices, how they are implemented, and the various filesystems used with them.

29.1 What are MTD Devices?

Memory Technology Devices (**MTD**) are **flash** memory devices. They are often used in various embedded devices.

Such a device may have all of its memory in flash (which functions like a hard disk in that its values are preserved upon power off) but often it will also have normal RAM of some type.

Flash memory is a high-speed EEPROM where data is programmed (and erased) in **blocks**, rather than byte by byte as in normal EEPROM.

MTD devices are neither character or block in type; in particular they distinguish between write and erase operations, which block devices don't.

Normal filesystems are generally not appropriate for use with flash devices for a number of reasons, which we'll detail later, so special filesystems have been designed.

The **Execute in Place**, or **XIP**, method, in which the CPU maps pages of memory from the flash-residing application directly to its virtual address space, with copying of pages to RAM first, can be useful in embedded devices.

Some useful references:

Table 29.1: **MTD links**

http://www.linux-mtd.infradead.org	The main web site for **Linux MTD** development.
http://www.linuxdevices.com/articles/AT7478621147.html	A white paper by Cliff Brake and Jeff Sutherland about using flash in embedded **Linux** systems.

29.2 NAND vs. NOR

There are two basic kinds of flash memory: **NOR** and **NAND**.

NOR flash devices are the older variety with these features:

- A linear addressed device, with individual data and address lines; just like **DRAM**.
- Addressed can be directly mapped in the CPU's address space and accessed like **ROM**.
- Programming and erase speeds are respectable; erases are slower than programs.
- Function like **RAM**, access is random.
- The number of erase cycles is limited, about 100,000 or so.
- Recent development of **MLC** (multi-level cell) techniques, in which two bits of memory can be stored per cell, have boosted density and reduced manufacturing costs per unit of memory, although it may come at the cost of reduced performance.
- Traditionally these devices have been associated with **code** storage.

NAND flash devices are newer, and have these features:

- Addressing is non-linear; data and commands are multiplexed onto 8 I/O lines. Thus, device drivers are more complex.
- Access is sequential.

- Densities are much higher than with **NOR** devices, and the speed is an order of magnitude faster.
- Bad blocks can be a problem; **NAND** devices may ship with them, but at any rate, blocks will fail with time, and thus device drivers have to do bad block management.
- Traditionally these devices have been associated with **data** storage.

Since 1999 **NAND** has grown from about one tenth of the total flash market to most of it.

Regardless of which method the underlying flash device uses, **Linux** can use the same basic methods to access it.

Here is a table from **http://www.linux-mtd.infradead.org/doc/nand.html** documenting some of the differences between **NAND** and **NOR** devices:

Table 29.2: **NOR and NAND device features**

	NOR	**NAND**
Interface	Bus	I/O
Cell Size	Large	Small
Cell Cost	High	Low
Read Time	Fast	Slow
Program Time (single byte)	Fast	Slow
Program Time (multi byte)	Slow	Fast
Erase Time	Slow	Fast
Power consumption	High	Low, but requires additional RAM
Can execute code	Yes	No, but newer chips can execute a small loader out of the first page
Bit twiddling	nearly unrestricted	1-3 times, also known as "partial page program restriction"
Bad blocks at ship time	No	Allowed

29.3 Driver and User Modules

The **MTD** subsystem in **Linux** uses a layered approach, in which the lower hardware device driver layer is ignorant of filesystems and storage formats, and need only have simple entry points for methods like **read, write,** and **erase**. Likewise, the upper layer is ignorant of the underlying hardware but handles all interaction with user-space.

Thus, there are two kinds of **modules** comprise the **MTD** subsystem; **user** and **driver**. These may or may not be actual kernel modules; they can be built-in.

User modules provide a high level interface to user-space, while **Driver** modules provide the raw access to the flash devices.

Currently implemented **User** modules include:

- **Raw character:** direct byte by byte access, needed to construct a filesystem or raw storage.
- **Raw block:** used to put normal filesystems on flash. Whole flash blocks are cached in RAM.
- **FTL, NFTL:** (Flash Translation Layer Filesystem)
- **Microsoft Flash Filing System:** Read-only for now.
- **Journalling Flash File System (JFFS2):** Full read/write, compressed journalling filesystem.

29.4 Flash Filesystems

Filesystems for flash devices pose some important challenges:

- Block sizes can be relatively large (64 KB to 256 KB). Under present **Linux** implementations, a block device filesystem can not have a block size bigger than a page frame of memory (4 KB on **x86** and many other platforms.)
- **NOR** flash has a finite limit to the number of erase cycles per block; typically about 100,000. It is important to use all parts of the device equally.
- There may be **bad blocks** which must to be locked out.
- Flash memory is expensive, so compressed filesystems are attractive.
- Journalling is important enhancement; it shortens the power-down procedure.
- Execution in place is often needed in embedded systems, but it is orthogonal to compression.

A number of different filesystems have been used for flash devices, and let's consider each in term.

initrd (**Initial Ram Disk**) was originally developed for use on floppy based systems, and then later to load a basic operating system which could then load essential drivers, such as in the case of **SCSI** systems.

When used with flash memory, **initrd** begins by copying a compressed kernel from flash to RAM and then executes the copy, which in turn decompresses the **initrd** image and mounts it using the ramdisk driver.

The disadvantages are fixed size, which is wasteful, and loss of changes upon reboot. Better approaches are now available.

cramfs is a compressed read only filesystem, where the compression is done at the unit of pages. An image is placed in flash and important system files are placed there. The filesystem is made with the **mkcramfs** program, and can be checked with the **cramfsck** utility, the sources of which can be found in **/usr/src/linux/scripts/cramfs/**.

ramfs is a dynamically-sized ramdisk, used in a flash filesystem to store frequently modified or temporary data.

jffs and its descendant **jffs2** are complete read/write, compressed, journalling filesystems. **jffs** was originally developed by Axis Communications in Sweden (**http://developer.axis.com/software /jffs**), and had no compression. **jffs2** provides compression and is developed by a team led by David Woodhouse (**http://sources.redhat.com/jffs2**).

The **jffs2** filesystem consists of a list of nodes (log entries) containing file information. When the filesystem is mounted the entire log is scanned to figure out how to put together files.

Nodes are written to flash sequentially from the first block on, and when the end is reached, the beginning is looped over. This spreads out access over the device; i.e., it provides **wear-leveling**.

Note that more than one filesystem can be used on a flash device. For instance read-only material can be put in a **cramfs** filesystem, frequently changing data can be placed on **ramfs**, and anything requiring read/write access and preservation across reboot can be put on **jffs2**.

29.5 Labs

Lab 1: Emulating MTD in memory

Even if you don't have any **MTD** devices on your system, you can emulate them in memory, using some built-in kernel features.

First you'll have to make sure you have all the right facilities built into the kernel. Go to the kernel source directory, run `make xconfig` and turn on the appropriate **MTD** options, as well as including the **jffs2** filesystem.

The important ones here are: under **MTD**, turn on *Memory Technology Device Support*, pick a level of debugging (3 should show all), turn on *Direct char device access...*, etc. Also turn on *Test driver using RAM* and *MTD emulation using block device*. By default you'll get a disk of 4 MB with 128 KB erase block size. Under *Filesystems*, turn on **JFFS(2)** and pick a verbosity level.

If you have done everything as modules you ***may*** get away without a reboot, as long as you run `depmod`. At any rate, recompile, reboot, etc., into the kernel that now includes **MTD** and **JFFS2**.

First we'll test the character emulation interface. To do this you have to make sure you create the device node:

```
mknod -m 666 /dev/mtd0 c 90 0
```

Before or after this, you'll have to make sure to do

```
modprobe mtdram total_size=2048 erase_size=8
```

(or leave out the options to get the default values you compiled into the kernel.) You won't have to run modprobe if you haven't done this as modules.

You can now use this as a raw character ram disk, reading and writing to it. Experiment using **dd, cat, echo**, etc.

Lab 2: Working with the jffs2 filesystem and the MTD block interface.

In order to place a **jffs2** filesystem on a **MTD** device it is easiest to first make a filesystem **image** on another filesystem, and then copy it over. To do this you must have the **mkfs.jffs2** utility, which you can download in source or binary form from **http://sources.redhat.com/jffs2**.

You'll need to do

```
modprobe mtdblock
```

if you haven't built this into the kernel.

You'll also have to make the proper device node:

```
mknod -m 666 /dev/mtdblock0 b 31 0
```

Populate a directory tree (say ./dir_tree) with some files and subdirectories; the total size should be less then or equal to the size of **MTD** ram disk. Then put a filesystem on it **and** copy it over to the **MTD** block device emulator with:

```
mkfs.jffs2 -d ./dir_tree -o /dev/mtdblock0
```

(You may want to separate out these steps so you can keep the initial filesystem image; i.e., do something like

```
mkfs.jffs2 -d ./dir_tree -o jfs.image
dd if=jfs.image of=/dev/mtdblock0
```

Now you can mount the filesystem and play with it to your heart's content:

```
mkdir ./mnt_jffs2
mount -t jffs2 /dev/mtdblock0 ./mnt_jffs2
```

Note that you can change the contents of the filesystem as you would like, but the updates will be lost when you unload the **MTD** modules or reboot. However, you can copy the contents to an image file and save that for a restore.

Note that if you have turned on some verbosity you will see messages like

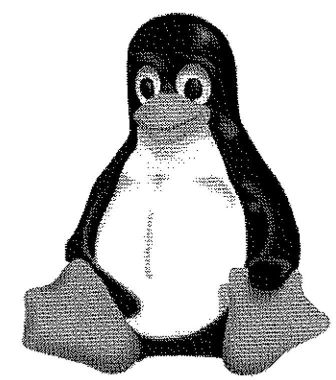

```
Feb 20 09:02:48 p3 kernel: ram_read(pos:520192, len:4096)
Feb 20 09:02:48 p3 kernel: ram_write(pos:393216, len:12)
Feb 20 09:02:48 p3 kernel: ram_read(pos:262144, len:4096)
Feb 20 09:02:48 p3 kernel: ram_read(pos:266240, len:4096)
Feb 20 09:02:48 p3 kernel: ram_read(pos:270336, len:4096)
```

and other such diagnostic information which will help you examine what is going on in the disk.

Notice that because **jffs2** is a compressed filesystem, you can accommodate much more than nominal size, depending on your actual contents.

Chapter 30

Power Management



We'll discuss how power management is done under **Linux**, using either the **APM** or **ACPI** protocols. We'll consider the possible states the system can be in, and what the power management functions are.

30.1 Power Management

Power management is handled by callback functions that are registered as part of device loading. The addresses of these functions are supplied as function pointers in appropriate structures such as `pci_driver`, `usb_driver`, etc.

At a lower level the device_driver structure will then contain pointers to these functions:

```
struct device_driver {
    ....
  int   (*probe)    (struct device * dev);
  int   (*remove)   (struct device * dev);
  void  (*shutdown) (struct device * dev);
  int   (*suspend)  (struct device * dev, pm_message_t state);
  int   (*resume)   (struct device * dev);
};
```

Many drivers are written with only some or even none of these functions supplied. This may be because the hardware has no advanced power capabilities.

More often it is because in the rush to getting a working device driver operable, the power management callback functions are put on the back burner and seen as an enhancement or optimization to be done later. Unfortunately they are not coded properly or never written.

There has been considerable frustration expressed about this in the kernel developer community, and even attempts to deny incorporation of drivers that fail to supply these callback functions.

The **lesswatts** project (**http://www.lesswatts.org**) contains a lot of information on tools, documentation, and methods for getting **Linux** to take better control of power management and reduce overall power consumption.

30.2 APM and ACPI

Virtually all **x86** mother boards will support either **APM** (Advanced Power Management), or the more recent **ACPI** (Advanced Configuration and Power Interface.) Other architectures such as **x86_64** also have the **ACPI** interface.

The big difference is that **APM** for the most part leaves power management in the hands of the system **BIOS**, and provides an interface to access it. The newer **ACPI** standard puts the power management directly in the hands of the operating system, which leads to far more flexibility and control. It also leads to much more direct control of peripherals.

If both **APM** and **ACPI** are turned on in the kernel, and if the system is compatible with **ACPI**, it will override and disable **APM**. You can not mix and match features of both as they will corrupt each other.

Both **APM** and **ACPI** require the use of user-space daemons (**apmd** and **acpid**) which should be started during the system boot by the usual startup scripts. If the system doesn't support the interface, the daemons will quit peacefully.

No matter which power management facility you use, **Linux** uses the same functions in the device drivers which simplifies things considerably. The actual interface is (fortunately) not very complicated.

30.3 System Power States

Under **APM** there are five states the system can be in:

Table 30.1: **APM power states**

State	Meaning
Full On	Default mode. No power management. All devices are on.
APM Enabled	System does work; some unused devices may not be powered. CPU clock may be slowed or stopped.
APM Standby	Enters this state after short period of inactivity; recovery to the Enabled state should appear instantaneous. Most devices in a low power mode. CPU clock may be slowed or stopped.
APM Suspend	Enters this state after long period of inactivity; recover to the Enabled state takes a longer period of time. System is in a low power state with maximum power savings, with most power managed devices powered off. CPU clock is stopped. System may go into **hibernation**, a special implementation which saves parameters.
Off	System off. All power off. Nothing saved.

Under **ACPI** there is a similar delineation, with an additional level of detail. The correspondences with **APM** are pretty transparent. With **G** standing for global, and **S** for sleeping, The states are:

Table 30.2: **ACPI power states**

State	Value	Meaning
Mechanical Off	**G3**	Power consumption is zero except for the real-time clock.
Soft Off	**G2/S5**	Minimal power used; no user or system code running. Takes a long time to go back to the Working state, and a restart has to be done to get there.
Sleeping	**G1**	Small amount of power used, user code not executed, some system code running but the device appears off; i.e., no display etc. Not a high latency to return to the Working state, but it can depend on how it was put to sleep. The operating system does not require a reboot to get going.

Working	**G0**	System using full power, running user and system code. Peripherals can have their power state dynamically modified and the user can modify power consumption characteristics.
Non-Volatile Sleep	**S4**	A special sleep state that lets context be saved and restored when power is cut off to the motherboard. (Hibernation). This storage of state is in non-volatile storage. This state is really a sub-state of the Sleeping state.

Under **ACPI** there are also a bunch of **Device Power State Definitions**. They range from **D0** to **D3**. All devices must have **DO** and **D3** modes defined, but **D1** and **D2** are optional.

Table 30.3: **ACPI device power states**

Value	Meaning
D3	Off. No power to the device. Context is lost; operating system must reinitialize upon re-powering.
D2	Meaning depends on the class of device. Saves more power than **D1**. Device may lose some of its context.
D1	Meaning depends on the class of the device. Saves more context than **D2**.
D0	Full power to the device, which is completely active, retaining all context.

30.4 Callback Functions

The following functions are part of the **ACPI** interface. If **ACPI** is not available, and **APM** is being used some of these functions become no-ops. Drivers do not have to be specifically aware of which scheme is being used; but they do have to be able to handle suspend or resume requests.

For **PCI** devices, power management facilities should be incorporated through functions pointed to by elements of the `pci_driver` data structure associated with the device:

```
struct pci_driver {
   struct list_head node;
   char *name;
```



```
    const struct pci_device_id *id_table;
    int  (*probe)(struct pci_dev *dev, const struct pci_device_id *id);
    void (*remove)(struct pci_dev *dev);
    int  (*suspend)(struct pci_dev *dev, pm_message_t state);
    int  (*resume)(struct pci_dev *dev);
    int  (*enable_wake)(struct pci_dev *dev, pci_power_t state, int enable);
    void (*shutdown)(struct pci_dev *dev);
    struct device_driver    driver;
    struct pci_dynids dynids;
};
```

Devices which reside on other buses have similar data structures which contain pointers to the necessary power management functions. For instance, for **USB** we have:

```
struct usb_driver {
   struct module *owner;
   const char *name;
   int (*probe) (struct usb_interface *intf, const struct usb_device_id *id);
   void (*disconnect) (struct usb_interface *intf);
   int (*ioctl) (struct usb_interface *intf, unsigned int code, void *buf);
   int (*suspend) (struct usb_interface *intf, pm_message_t message);
   int (*resume) (struct usb_interface *intf);
   const struct usb_device_id *id_table;
   struct device_driver driver;
};
```

Kernels before the 2.6 series contained a generic interface, whose main functions and data structures were

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

struct pm_dev *pm_register (pm_dev_t type, unsigned long id, pm_callback callback);
void pm_unregister (struct pm_dev *dev);
void pm_access     (struct pm_dev *dev);
void pm_dev_idle   (struct pm_dev *dev);
int (*pm_callback) (struct pm_dev *dev, pm_request_t rqst,
                    void *data);
struct pm_dev
{
  pm_dev_t       type;
  unsigned long  id;
  pm_callback    callback;
  void          *data;
  unsigned long  flags;
  int            state;
  int            prev_state;
  struct list_head entry;
};
```

This interface is now marked as **deprecated**, and should not be used in new code.

The newer device-tree based interface can handle geometric dependencies; e.g., one must turn off all **PCI** devices before turning off the **PCI** bus. Furthermore it is based on the new unified device model and is quite straightforward.

For a an excellent description of the old and new interfaces look at **http://tree.celinuxforum.org /CelfPubWiki/PmSubSystem**.

30.5 Labs

Lab 1: Monitoring Power Usage with powertop

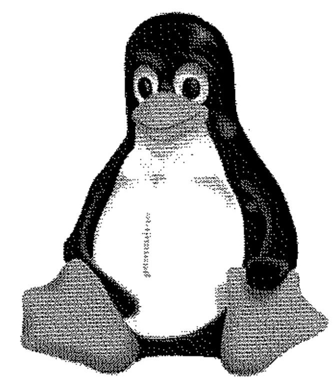
A tool for assessing power consumption can be obtained from **http://www.lesswatts.org/projects /powertop/**.

powertop can monitor how many times CPU's are being woken up every second, and help curtail unnecessary activity in order to save power. It can also keep track of total power consumption and suggest methods of reduction.

To make **powertop** work `CONFIG_TIMER_STATS` must be set in the kernel configuration.

Chapter 31

Notifiers



We'll discuss how the **Linux** kernel implements notifier callback chains so that interested parties can monitor various kernel resources and subsystems. We'll show how to create a notifier chain as well as how to register with a preexisiting one. We'll explain how to write callback functions and insert them in the relevant chain.

31.1 What are Notifiers?

Sometimes a particular piece of kernel code needs either to inform other parts of the kernel about an event of interest, or needs to be alerted to events that may be of interest to itself. While a number of methods of such notification have been employed in the past, the present **kernel notifier** API was introduced in the 2.6.17 kernel.

Examples of events which utilize notifiers include:

- Network device changes.
- CPU frequency changes.
- Memory hotplug events.
- **USB** hotplug events.
- Module loading/unloading.
- System reboots.

There are four kinds of notifier **chains** which can be used:

- **Blocking:** Callbacks are run in process context and are allowed to block.
- **Atomic:** Callbacks are run in interrupt/atomic context and are not allowed to block.
- **Raw:** Callbacks are unrestricted (as is registration and unregistration) but locking and protection must be explicitly provided by callers.
- **SRCU:** A form of blocking notifier that uses **Sleepable Read-Copy Update** instead of read/write semaphores for protection.

The blocking and atomic types are the ones used most often and for simplicity we'll restrict our discussion to these two types.

31.2 Data Structures

The important data structures for notifier chains are the `notifier_block` and the various `notifier_head` structures:

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

struct notifier_block {
   int (*notifier_call) (struct notifier_block *block, unsigned long event, void *data);
   struct notifier_block *next;
   int priority;
};

struct blocking_notifier_head {
        struct rw_semaphore rwsem;
        struct notifier_block *head;
};

struct atomic_notifier_head {
        spinlock_t lock;
        struct notifier_block *head;
};
```


The `notifier_call()` is the function to be called when something of interest occurs and it will receive `event` and a pointer to `data` when called.

The `next` element shows there will be a linked list of notifier functions, called in order of priority.

The `priority` data element works so that the **final** event called is the one with the highest priority (lowest value for the `priority` field); if the bit `NOTIFIER_STOP_MASK` is set in the callback function return value, any notifier can stop any further processing. Other return values are not confined, but the special values `NOTIFY_STOP` (everything is fine, don't call any more modifiers) and `NOTIFY_OK` (everything is fine, continue calling other callback functions) can be used.

31.3 Callbacks and Notifications

Registering and unregistering callback functions is done with:

```
int atomic_notifier_chain_register (struct atomic_notifier_head *nh,
                                    struct notifier_block *nb);
int blocking_notifier_chain_register (struct blocking_notifier_head *nh,
                                      struct notifier_block *nb);

int atomic_notifier_chain_unregister (struct atomic_notifier_head *nh,
                                      struct notifier_block *nb);
int blocking_notifier_chain_unregister (struct blocking_notifier_head *nh,
                                        struct notifier_block *nb);
```

These functions tell the system to call the function specified in the `notifier_block` function whenever traversing the linked list in the notifier head, which may be one you created or which previously existed.

Signalling an event to the appropriate notifier chain is done with:

```
int blocking_notifier_call_chain (struct blocking_notifier_head *nh, unsigned long event, void *dat
int atomic_notifier_call_chain   (struct atomic_notifier_head *nh, unsigned long event, void *data)
```

where the `event` is specified and a pointer to `data` can be passed.

Pre-existing notifier chains generally follow the convention of defining registration/unregistration functions as:

```
void XXX_register_notifier  (struct notifier_block *nb);
void XXX_unregister_NOTIFIER(struct notifier_block *nb);
```

where `XXX` specifies the notifier; examples include `usb`, `reboot`, `cpu_notifier`, `crypto`, `oom`, and `netdevice`. Occasionally the `XXX` and the `register`, `unregister` elements in the names are swapped.

31.4 Creating Notifier Chains

Creating blocking and atomic notifier chains can be done in the either by doing:

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

BLOCKING_NOTIFIER_HEAD(notifier_name);
ATOMIC_NOTIFIER_HEAD(notifier_name);
```

or

```
struct blocking_notifier_head notifier_name;
BLOCKING_INIT_NOTIFIER_HEAD(notifier_name);

struct atomic_notifier_head notifier_name;
ATOMIC_INIT_NOTIFIER_HEAD(notifier_name);
```

which create the appropriate **notifier_head** structures:

31.5 Labs

Lab 1: Joining the USB Notifier Chain

To register and unregister with the already existing notifier chain for hot-plugging of **USB** devices, use the exported functions:

```
void usb_register_notify   (struct notifier_block *nb);
void usb_unregister_notify (struct notifier_block *nb);
```

You should be able to trigger events by plugging and unplugging a **USB** device, such as a mouse, pendrive, or keyboard.

Print out the event that triggers your callback function. (Note that definitions of events can be found in **/usr/src/linux/include/linux/usb.h**.).

Lab 2: Installing and Using a Notifier Chain

Write a brief module that implements its own notifier chain.

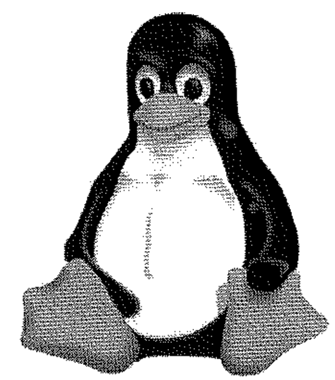
The module should register the chain upon insertion and unregister upon removal.

The callback function should be called at least twice, with different event values, which should be printed out.

You may want to make use of the data pointer, modifying the contents in the callback function.

Chapter 32

CPU Frequency Scaling



We'll discuss how **Linux** can adjust CPU frequency dynamically. We'll show how to register notifier callback functions to monitor or influence changes in policy and speed. We'll also consider how to write hardware-specific drivers and governors to control the policy.

32.1 What is Frequency and Voltage Scaling?

Frequency scaling permits dynamic adjustments to a CPU's clock speed (and voltage) according to system load.

The goal is to lower power consumption and heat generation; in the case of laptops this extends battery life and in the case of desktops and servers can prolong hardware life, lower electric bills and cooling needs, and along the way help save the planet.

Obviously, exactly how this might be done is very much hardware-dependent. There are three general possibilities:

- It may be impossible to have any any clock speed adjustment; this is likely on old CPUs.
- There may be only two states, high and low speed. For instance a laptop might have one high speed state to use when on AC power, and one low speed state to use when on battery power.
- There will be a finite number of intermediate states in between the lowest and highest speed.

When it comes to frequency scaling there is a clear separation between **mechanism** and **policy**.

The **mechanism** is encapsulated in the CPU frequency scaling device driver. This has to be written for each kind of hardware and is responsible for dealing with the actual changes.

The **policy** is determined by the CPU frequency scaling **governor**. This can be:

- **Performance:** The frequency is set to the highest possible.
- **Powersave:** The frequency is set to the lowest possible.
- **Userspace:** The frequency can be set manually or dynamically by a user-space program.
- **Ondemand:** The frequency is adjusted dynamically based on periodic polling of the CPU load. The CPU must be capable of fast frequency shifting.
- **Conservative:** Similar to ondemand, but the frequency is gradually increased or decreased rather than being done all in one step. This is more suitable for laptops than desktops.

Changes in policy as well as changes in frequency are broadcast through the use of **notifier** chains, and any kernel component can register call back functions that are invoked when there are changes.

The frequency scaling implementation in **Linux** is written in a uniform manner for all architectures, and it is quite modular especially as concerns implementing new governing policies.

32.2 Notifiers

There are two kind of CPU frequency notifiers; **transition** and **policy**.

Registering and unregistering a callback function is done with:

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

int cpufreq_register_notifier   (struct notifier_block *nb, unsigned int list);
int cpufreq_unregister_notifier (struct notifier_block *nb, unsigned int list);
```

where `list` can be either `CPUFREQ_TRANSITION_NOTIFIER` or `CPUFREQ_POLICY_NOTIFIER`. Remember the prototype of a notifier block structure is:


```
2.6.31:  50 struct notifier_block {
2.6.31:  51     int (*notifier_call)(struct notifier_block *, unsigned long, void *);
2.6.31:  52     struct notifier_block *next;
2.6.31:  53     int priority;
2.6.31:  54 };
```

Transition notifier callback functions are called twice every time the frequency changes, with values for the `event` argument being `CPUFREQ_PRECHANGE` and `CPUFREQ_POSTCHANGE`. They can also be called with the values `CPUFREQ_RESUMECHANGE` and `CPUFREQ_SUSPENDCHANGE` if the CPU changes its frequency while the system is suspended or resuming.

The third argument to the callback function for a transition notifier points to a data structure of type:

```
2.6.31: 123 struct cpufreq_freqs {
2.6.31: 124     unsigned int cpu;       /* cpu nr */
2.6.31: 125     unsigned int old;
2.6.31: 126     unsigned int new;
2.6.31: 127     u8 flags;               /* flags of cpufreq_driver, see below. */
2.6.31: 128 };
```

Policy notifier callback functions are called three times when a policy is set. First they are called with `event` being `CPUFREQ_ADJUST`; any notifier can change the limits if they perceive a need, for instance to avoid thermal problems are hardware limitations.

The second time `event` has the value `CPUFREQ_INCOMPATIBLE`. Changes should only be made to avoid hardware failure.

The third time `event` is `CPUFREQ_NOTIFY`. All notifiers are informed of the new policy and if two hardware drivers fail to agree before this stage, incompatible hardware is shut down.

The third argument to the callback function points to a structure of type:

```
2.6.31:  82 struct cpufreq_policy {
2.6.31:  83     cpumask_var_t           cpus;   /* CPUs requiring sw coordination */
2.6.31:  84     cpumask_var_t           related_cpus; /* CPUs with any coordination */
2.6.31:  85     unsigned int            shared_type; /* ANY or ALL affected CPUs
2.6.31:  86                                                 should set cpufreq */
2.6.31:  87     unsigned int            cpu;    /* cpu nr of registered CPU */
2.6.31:  88     struct cpufreq_cpuinfo  cpuinfo;/* see above */
2.6.31:  89
2.6.31:  90     unsigned int            min;    /* in kHz */
2.6.31:  91     unsigned int            max;    /* in kHz */
2.6.31:  92     unsigned int            cur;    /* in kHz, only needed if cpufreq
2.6.31:  93                                      * governors are used */
2.6.31:  94         unsigned int                policy; /* see above */
2.6.31:  95     struct cpufreq_governor *governor; /* see below */
2.6.31:  96
2.6.31:  97     struct work_struct      update; /* if update_policy() needs to be
2.6.31:  98                                      * called, but you're in IRQ context */
2.6.31:  99
2.6.31: 100     struct cpufreq_real_policy      user_policy;
2.6.31: 101
2.6.31: 102     struct kobject          kobj;
```

```
2.6.31: 103     struct completion       kobj_unregister;
2.6.31: 104 };
```

The main fields here are the CPU number, the minimum and maximum frequencies, the current frequency, and the policy,

32.3 Drivers

A frequency scaling driver first has to check to see whether it is appropriate for the kernel, CPU and chipset. In the initialization routine, there should first be some kind of check and if things are not suitable the driver should not continue to load.

The registration/unregistration of a CPU frequency driver is done with:

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

int cpufreq_register_driver   (struct cpufreq_driver *driver_data);
int cpufreq_unregister_driver (struct cpufreq_driver *driver_data);
```

where the cpufreq_driver structure should be filled out first, and looks like:

```
2.6.31: 211 struct cpufreq_driver {
2.6.31: 212     struct module           *owner;
2.6.31: 213     char                    name[CPUFREQ_NAME_LEN];
2.6.31: 214     u8                      flags;
2.6.31: 215
2.6.31: 216     /* needed by all drivers */
2.6.31: 217     int     (*init)         (struct cpufreq_policy *policy);
2.6.31: 218     int     (*verify)       (struct cpufreq_policy *policy);
2.6.31: 219
2.6.31: 220     /* define one out of two */
2.6.31: 221     int     (*setpolicy)    (struct cpufreq_policy *policy);
2.6.31: 222     int     (*target)       (struct cpufreq_policy *policy,
2.6.31: 223                              unsigned int target_freq,
2.6.31: 224                              unsigned int relation);
2.6.31: 225
2.6.31: 226     /* should be defined, if possible */
2.6.31: 227     unsigned int    (*get)  (unsigned int cpu);
2.6.31: 228
2.6.31: 229     /* optional */
2.6.31: 230     unsigned int (*getavg)  (struct cpufreq_policy *policy,
2.6.31: 231                              unsigned int cpu);
2.6.31: 232
2.6.31: 233     int     (*exit)         (struct cpufreq_policy *policy);
2.6.31: 234     int     (*suspend)      (struct cpufreq_policy *policy, pm_message_t pmsg);
2.6.31: 235     int     (*resume)       (struct cpufreq_policy *policy);
2.6.31: 236     struct freq_attr        **attr;
2.6.31: 237 };
```

The init() function points to the initialization function which will be run per-CPU, which has a cpufreq_policy structure as its argument. This function has to activate CPU frequency support,

fill in the fields cpuinfo.min_freq and cpuinfo.max_freq (in kHz), cpinfo.transition_latency (the time required to switch between two frequencies), and various other fields.

The verify() function has to validate any new policy to be set. It can make use of a table of permissible values.

One uses either the setpolicy() or target() function pointer, but not both. If the CPU can be set to only one frequency, the target() call is used; otherwise the setpolicy() function is used.

We won't discuss this further as the details get very hardware-dependent. Once can find decent information about the drivers for various CPUs and chip sets by looking at **/usr/src/linux/Documentation/cpu-freq**.

32.4 Governors

The powersave and performance governors are built-in to the **Linux** kernel, and merely set the CPU frequency to the lowest and highest possible frequencies.

More complicated **governors** are possible and are provided with the mainline kernel, and it is possible to add other governor implementations, either as built-in or with a kernel module.

One has to register/unregister the governor with:

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

int cpufreq_register_governor    (struct cpufreq_governor *governor);
void cpufreq_unregister_governor (struct cpufreq_governor *governor);
```

where the cpufreq_governor structure has to be filled out first and looks like:

```
2.6.31: 165 struct cpufreq_governor {
2.6.31: 166     char    name[CPUFREQ_NAME_LEN];
2.6.31: 167     int     (*governor)     (struct cpufreq_policy *policy,
2.6.31: 168                              unsigned int event);
2.6.31: 169     ssize_t (*show_setspeed)        (struct cpufreq_policy *policy,
2.6.31: 170                                      char *buf);
2.6.31: 171     int     (*store_setspeed)       (struct cpufreq_policy *policy,
2.6.31: 172                                      unsigned int freq);
2.6.31: 173     unsigned int max_transition_latency; /* HW must be able to switch to
2.6.31: 174                     next freq faster than this value in nano secs or we
2.6.31: 175                     will fallback to performance governor */
2.6.31: 176     struct list_head        governor_list;
2.6.31: 177     struct module           *owner;
2.6.31: 178 };
```

where the main entry which must be filled in is the pointer to the governor() callback function, which can be called with one of the three following values for event:

- CPUFREQ_GOV_START: Start operating for this CPU.
- CPUFREQ_GOV_STOP: End operating for this CPU.

- CPUFREQ_GOV_LIMITS: The maximum and minimum limits have changed for this CPU.

The callback function can call the CPU frequency driver using:

```
int cpufreq_driver_target(struct cpufreq_policy *policy, unsigned int target_freq,
                          unsigned int relation);
```

where relation can be CPUFREQ_REL_L (try to select a new frequency higher than or equal to the target frequency) or CPUFREQ_REL_H (try to select a new frequency lower than or equal to the target frequency.)

32.5 Labs

Lab 1: CPU Frequency Notifiers

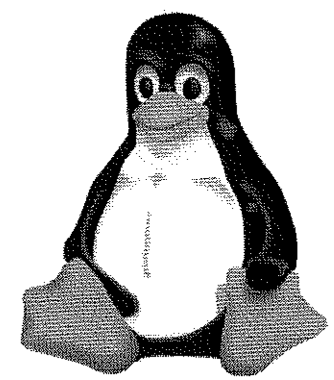
Write a module that registers callback functions for the CPU frequency transition and policy notifier chains.

Print out what event is causing the callback, and some information from the data structures delivered to the callback functions.

You can test this by echoing values to some of the entries in **/sys/devices/system/cpu/cpu0 /cpufreq**. Even easier you can add the **CPU Frequency Scaling Monitor** applet to your taskbar, and easily switch governors and frequencies.

Chapter 33

Asynchronous I/O



We will discuss asynchronous I/O, the functional interface for it, and methods of implementation under **Linux**.

33.1 What is Asynchronous I/O?

Normally all I/O operations are performed **synchronously**; an application will block until the read or write is completed, successfully or unsuccessfully.

Note that this doesn't mean all pending writes will be flushed to disk immediately, only that interaction with the virtual file system has been completed.

But what if I/O requests could be queued up, and program execution continued in parallel with completion of the I/O request? This can be particularly useful on SMP systems and when using **DMA**, which does not involve the CPU. When this is done, it is called **asynchronous I/O**, or **AIO**.

In order for this to work properly, there has to be notification when the I/O request is complete and code must contain synchronization points, or completion barriers.

In addition, policies must be set as to whether queued requests are serialized, especially when they refer to the same file descriptors.

You can request asynchronous I/O by using the ASYNC flag when opening a file, or the -o async option to the **mount** command, but this won't give true asynchronous I/O.

33.2 The Posix Asynchronous I/O API

The POSIX 1.b standard defines a basic data structure, the aiocb, which stands for **AIO control block**, and a set of basic functions that can be performed on it. These are defined and prototyped in **/usr/include/aio.h** and are provided with **glibc**:

The data structure looks like:

```
struct aiocb
{
  int aio_fildes;                /* File desriptor.  */
  int aio_lio_opcode;            /* Operation to be performed.  */
  int aio_reqprio;               /* Request priority offset.  */
  volatile void *aio_buf;        /* Location of buffer.  */
  size_t aio_nbytes;             /* Length of transfer.  */
  struct sigevent aio_sigevent;  /* Signal number and value.  */
  __off64_t aio_offset;          /* File offset
  ...
};
```

where we have omitted the purely internal members of the data structure.

aio_fildes can be any valid file descriptor, but it must permit use of the lseek() call.

aio_lio_opcode is used by the lio_listio() function and stores information about the type of operation to be performed.

aio_reqprio can be used to control scheduling priorities.

aio_buf points to the buffer where the data is to be written to or read from.

aio_nbytes is the length of the buffer.

aio_sigevent controls what if any signal is sent to the calling process when the operation completes.

aio_offset gives the offset into the file where the I/O should be performed; this is necessary because doing I/O operations in parallel voids the concept of a current position.

The basic functions are actually not part of **glibc** proper, but are part of another library, **librt**. These functions are:

```
#include <aio.h>

void aio_init (const struct aioinit *init);
int aio_read  (struct aiocb *cb);
```



```
int aio_write (struct aiocb *cb);
int iol_listio(int mode, struct aiocb *const cblist[], int nent, struct sigevent *sig);
int aio_error (const struct aiocb *cb);
ssize_t aio_return (const struct aiocb *cb);
int aio_fsync (int op, struct aiocb *cb);
int aio_suspend (const struct aiocb *cb, const cblist[], int nent,
                 const struct timespec *timeout);
int aio_cancel (int fd, struct aiocb *cb);
```

A typical code fragment might have:

```
struct aiocb *cb = malloc (sizeof struct aiocb);
char *buf = malloc (nbytes);
....
fd = open (....);
cb->aio_filedes = fd;
cb->aio_nbytes = nbytes;
cb->aio_offset = offset;
cb->aio_buf = buf;
....
rc = aio_read (cb);
....
while (aio_error (cb) == EINPROGRESS){};
....
```

where we use the aio_error() function call to wait for completion of pending requests on the control block.

33.3 Linux Implementation

The original **AIO** implementation for **Linux** was done by **glibc** completely in user-space. A thread was launched for each file descriptor for which there were pending **AIO** requests.

This approach is costly, however, if there are large numbers, even thousands, of such requests; true support within the kernel can lead to far better performance. Thus **glibc** also permits the important parts of the implementation to be passed off to the kernel and done more efficiently in kernel-space.

The 2.6 kernel contains full kernel support for **AIO**; in fact **all** I/O is really be done through the asynchronous method, with normal I/O being the result if certain flags are not set.

A document describing the details of this implementation can be found at **http://lse.sourceforge.net/io/aionotes.txt**.

Block and network device drivers already fully take advantage of the asynchronous implementation. Character drivers (which rarely require asynchronism), however, need to be modified specifically to take advantage of the new facility. This means supplying new functions in the file_operations jump table data structure, for:

```
ssize_t (*aio_read) (struct kiocb *iocb, const struct iovec *iov, unsigned long niov,
                     loff_t pos);
```

```
ssize_t (*aio_write)(struct kiocb *iocb, const struct iovec *iov, unsigned long niov,
                     loff_t pos);
int (*aio_fsync) (struct kiocb *, int datasync);
```

As of this writing **glibc** still does not take advantage of full kernel support for **AIO** for **Linux**. Thus, even if you put such entry points in your driver, they'll never get hit. (Actually if you don't have normal read and write entry points the kernel will call the asynchronous ones, if you want to test them.)

However, there is a native user-space **API** in **Linux** with new system calls that can be used efficiently; it just isn't portable. To use this you have to have the **libaio** package installed, and if you want to compile code using it you have to have the **libaio-devel** package installed. Your code will have to include the header file **/usr/include/libaio.h**. The basic functions are:

```
#include <libaio.h>

long io_setup (unsigned nr_events, aio_context_t *ctxp);
long io_submit (aio_context_t ctx_id, long nr, struct iocb **iocbpp);
long io_getevents (aio_context_t ctx_id, long min_nr, long nr, struct io_event *events,
                   struct timespec *timeout);
long io_destroy (aio_context_t ctx);
long io_cancel (aio_context_t ctx_id, struct iocb *iocb, struct io_event *result);
```

These functions all have **man** pages so we won't describe them completely.

Before any I/O work can be done, a **context** has to be set up to which any queued calls belong; otherwise the kernel may not know who they are associated with. This is done with the call to `io_setup()`, where the context **must** be initialized; e.g.,

```
io_context_t ctx = 0;
rc = io_setup (maxevents, &ctx);
```

where `maxevents` is the largest number of asynchronous events that can be received. The handle returned is then passed as an argument in the other functions. The function `io_destroy()` will wipe out the context when you are finished.

The `io_submit()` function is used to submit asynchronous requests, which have their **iocb** structures properly set up.

The `io_getevents()` function is used to check the status, and `io_cancel()` can be used to try and cancel a pending request. (**Note**; the `events` argument must point to an **array** of structures at least as large as the maximum number of events you are looking at. The documentation is not clear about this and missing it is a good way to get segmentation faults.)

The control block structure itself is given by:

```
struct iocb {
   void *data;   /* Return in the io completion event */
   unsigned key; /* For use in identifying io requests */

   short   aio_lio_opcode;
   short   aio_reqprio;
```



```
   int     aio_fildes;

   union {
      struct io_iocb_common      c;
      struct io_iocb_vector      v;
      struct io_iocb_poll      poll;
      struct io_iocb_sockaddr   saddr;
   } u;
};

struct io_iocb_poll {
   int events;
};   /* result code is the set of result flags or -'ve errno */

struct io_iocb_sockaddr {
   struct sockaddr *addr;
   int      len;
};   /* result code is the length of the sockaddr, or -'ve errno */

struct io_iocb_common {
   void           *buf;
   unsigned long nbytes;
   long long     offset;
};   /* result code is the amount read or -'ve errno */

struct io_iocb_vector {
   const struct iovec *vec;
   int                nr;
   long long          offset;
};   /* result code is the amount read or -'ve errno */
```

While you can get by with just these functions and structures, it can be tedious to insert all the right values in the right places before submitting requests. There are a number of helper functions in the header file which will do most of the work for you. Unfortunately the **man** pages don't mention them. Two in particular you will definitely want to use are:

```
void io_prep_pread  (struct iocb *iocb, int fd, void *buf, size_t count, long long offset);
void io_prep_pwrite (struct iocb *iocb, int fd, void *buf, size_t count, long long offset);
```

After calling these functions in an obvious way, you can just call `io_submit()` to get your I/O going.

Eventually the **glibc** maintainers will get around to doing the wrapping necessary to permit the **Posix API** to be used. At that point, if portability is desired, the native **Linux** applications that use **AIO** should be portable in a straightforward way, if portability is required.

33.4 Labs

Lab 1: Adding Asynchronous Entry Points to a Character Driver

Take one of your earlier character drivers and add new entry points for `aio_read()` and `aio_write()`.

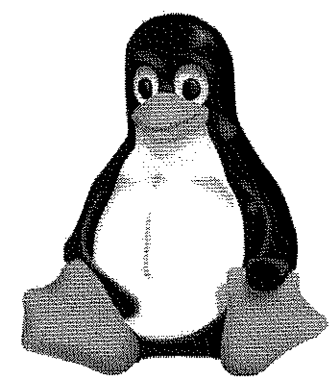
To test this you'll need to write a user-space program that uses the native **Linux API**. Have it send out a number of write and read requests and synchronize properly.

We also present a solution using the **Posix API** for the user application; note that this will never hit your driver unless you comment out the normal read and write entry points in which case the kernel will fall back on the asynchronous ones.

Make sure you compile by linking with the right libraries; use `-laio` for the **Linux API** and `-lrt` for the **Posix API**. (You can use both in either case as they don't conflict.)

Chapter 34

I/O Scheduling



We consider I/O scheduling, and the various algorithms **Linux** uses.

34.1 I/O Scheduling

The **I/O scheduler** provides the interface between the generic block layer and low-level physical device drivers. Both the VM and VFS layers submit I/O requests to block devices; it is the job of the I/O scheduling layer to prioritize and order these requests before they are given to the block devices.

Any I/O scheduling algorithm has to satisfy certain (sometimes conflicting) requirements:

- Hardware access times should be minimized; i.e., requests should be ordered according to physical location on the disk. This leads to an **elevator** scheme where requests are inserted in the pending queue in physical order.

- Requests should be merged to the extent possible to get a big a contiguous region as possible, which also minimizes disk access time.
- Requests should be satisfied with as low a latency as is feasible; indeed in some cases determinism (in the sense of deadlines) may be important.
- Write operations can usually wait to migrate from caches to disk without stalling processes. Read operations, however, almost always require a process to wait for completion before proceeding further. Thus favoring reads over writes leads to better parallelism and system responsiveness.
- Processes should share the I/O bandwidth in a fair, or at least consciously prioritized fashion; even it means some overall performance slowdown of the I/O layer, process throughput should not suffer inordinately.

Since these demands can be conflicting, different I/O schedulers may be appropriate for different workloads; e.g., a large database server vs. a desktop system. In order to provide flexibility, the 2.6 **Linux** kernel has an object oriented scheme, in which pointers to the various needed functions are supplied in a data structure, the particular one of which can be selected at run time. This is done with:

```
linux ...  elevator=[anticipatory|cfq|deadline|noop]
```

At least one of the I/O scheduling algorithms must be compiled into the kernel. We'll discuss each of the following in some detail:

- **Anticipatory Scheduling** (AS)
- **Completely Fair Queueing** (cfq)
- **Deadline Scheduling**
- **noop** (A simple scheme)

The default choice is a compile configuration option; for kernels before 2.6.18 it was was **AS**, while it is now **CFQ** but distributions may have a different preference.

It is possible to use different I/O schedulers for different devices. The choice can be made easily through the command line, or from within kernel code by calling the function

```
int elevator_init(struct request_queue *q, char *name);
```

For instance the generic block layer calls this function with `name` set equal to `NULL` so as to get the default, while the block tape device driver calls it with `name` set equal to `noop`.

34.2 Tunables

Each of the I/O schedulers exposes parameters which can be used to tune behaviour at run time. The parameters are accessed through the **sysfs** filesystem.

One can change the scheduler being used for a device:

```
$ cat /sys/block/sda/queue/scheduler

noop [anticipatory] deadline cfq

$ echo cfq > /sys/block/sda/queue/scheduler
$ cat /sys/block/sda/queue/scheduler

noop anticipatory deadline [cfq]
```

The actual tunables vary according to the particular I/O scheduler, and can be found under:

```
/sys/block/<device>/queue/iosched
```

For example:

```
$ ls -l /sys/block/sda/queue/iosched
total 0
-rw-r--r--  1 root root 4096 Mar 27 17:42 antic_expire
-r--r--r--  1 root root 4096 Mar 27 17:42 est_time
-rw-r--r--  1 root root 4096 Mar 27 17:42 read_batch_expire
-rw-r--r--  1 root root 4096 Mar 27 17:42 read_expire
-rw-r--r--  1 root root 4096 Mar 27 17:42 write_batch_expire
-rw-r--r--  1 root root 4096 Mar 27 17:42 write_expire
```

We'll discuss some of the tunables for the individual I/O schedulers.

34.3 noop I/O Scheduler

This simple scheduler focuses on disk utilization. For a given device, a single queue is maintained. For each request it is determined if the request can be merged (front or back) with any existing request. If not the request is inserted in the queue according to the starting block number.

In order to prevent the request from going stale, an aging algorithm determines how many times an I/O request may have been bypassed by newer requests, and above a threshold prompts the request to be satisfied.

The **noop** scheduler is particularly useful for non-disk based block devices (such as ram disks), as well as for advanced specialized hardware that has its own I/O scheduling software and caching, such as some RAID controllers. By letting the custom hardware/software combination make the decisions, this scheduler may actually outperform the more complex alternatives.

34.4 Deadline I/O Scheduler

The **deadline** I/O scheduler aggressively reorders requests with the simultaneous goals of improving overall performance and preventing large latencies for individual requests; i.e., limiting starvation.

With each and every request the kernel associates a **deadline**. Read requests get higher priority than write requests.

Five separate I/O queues are maintained:

- Two **sorted** lists are maintained, one for reading and one for writing, and arranged by starting block.
- Two **FIFO** lists are maintained, again one for reading and one for writing. These lists are sorted by submission time.
- A fifth queue contains the requests that are to be shoveled to the device driver itself. This is called the **dispatch** queue.

Exactly how the requests are peeled off the first four queues and placed on the fifth (dispatch queue) is where the art of the algorithm is.

Tunables

`read_expire:`

How long (in milliseconds) a read request is guaranteed to occur within. (Default = `HZ/2 = 500`)

`write_expire:`

How long (in milliseconds) a write request is guaranteed to occur within. (Default = `5 * HZ = 5000`)

`writes_starved:`

How many requests we should give preference to reads over writes. (Default = 2)

`fifo_batch:`

How many requests should be moved from the sorted scheduler list to the dispatch queue, when the deadlines have expired. (Default = 16)

`front_merges:`

Back merges are more common than front merges as a contiguous request usually continues to the next block. Setting this parameter to 0 disables front merges and can give a boost if you know they are unlikely to be needed. (Default = 1)

Some detailed documentation can be found at: **/usr/src/linux/Documentation/block/deadline-iosched.txt**.

34.5 Completely Fair Queue Scheduler

The **cfq** (Completely **Fair Queue**) method has the goal of equal spreading of I/O bandwidth among all processes submitting requests.

Theoretically each process has its own I/O queue, which work together with a dispatch queue which receives the actual requests on the way to the device. In practice the number of queues is fixed (at 64) and a hash process based on the process ID is used to select a queue when a request is submitted.

Dequeuing of requests is done round robin style on all the queues, each one of which works in FIFO order. Thus the work is spread out. To avoid excessive seeking operations, an entire round is selected, and then sorted into the dispatch queue before actual I/O requests are issued to the device.

Tunables

`quantum`

Maximum queue length in one round of service. (Default = 4);

`queued`

Minimum request allocation per queue. (Default = 8)

`fifo_expire_sync`

FIFO timeout for sync requests. (Default = `HZ/2`)

`fifo_expire_async`

FIFO timeout for async requests. (Default = `5 * HZ`)

`fifo_batch_expire`

Rate at which the FIFO's expire. (Default = `HZ/8`)

`back_seek_max`

Maximum backwards seek, in KB. (Default = `16K`)

`back_seek_penalty`

Penalty for a backwards seek. (Default = 2)

34.6 Anticipatory I/O Scheduler

The **anticipatory I/O scheduler** works off the observation that disk reads are often followed by other disk reads of nearby sectors.

When a request is made, a timer starts and I/O requests are not forwarded to the driver until it expires; i.e., the request queue is ***plugged*** momentarily. In the meantime, if another ***close*** request arrives it is serviced immediately.

The algorithm is adaptive in that is constantly adjusts its concept of ***close*** in accordance with the actual I/O request load.

When there are no more close requests, work continues with normal pending I/O requests.

The basic goal is to reduce the per-thread response time. This scheme is similar in implementation to the **deadline** scheduler, and can be consider a variation on it.

Tunables

`antic_expire:`

Maximum time (in milliseconds) to wait anticipating a good read (close to the most recently completed request) before giving up. (Default = `HZ/150 = 6`)

`read_expire:`

How long (in milliseconds) until a read request expires, as well as the interval between serving expired requests. (Default = `HZ/8 = 125`)

`read_batch_expire:`

How long (in milliseconds) a batch of reads gets before pending writes are served. Should be a multiple of `read_expire`. The higher the value, the more reading is favored over writing. (Default = HZ/2 = 500)

`write_expire` and `write_batch_expire:`

Serve the same functions for writes. (Defaults = `HZ/4 = 250, HZ/8 = 125`)

`est_time:`

Gives some statistics.

Some detailed documentation can be found at: **/usr/src/linux/Documentation/block/as-iosched.txt:**

34.7 Labs

Lab 1: Comparing I/O schedulers

Write a script (or program if you prefer) that cycles through available I/O schedulers on a hard disk and does a configurable number of parallel reads and writes of files of a configurable size. You'll probably want to test reads and writes as separate steps.

To test reads you'll want to make sure you're actually reading from disk and not from cached pages of memory; you can flush out the cache by doing

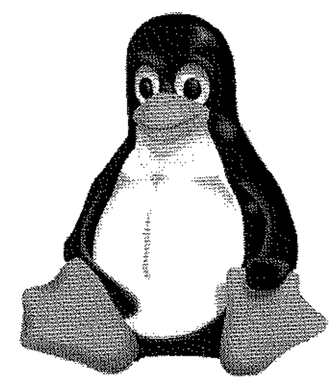
```
$ echo 3 > /proc/sys/vm/drop_caches
```

before doing the reads. You can **cat** into `/dev/null` to avoid writing to disk. To make sure all reads are complete before you get timing information, you can issue a **wait** command under the shell.

To test writes you can simply copy a file (which will be in cached memory after the first read) multiple times simultaneously. To make sure you wait for all writes to complete before you get timing information you can issue a **sync** call.

Chapter 35

Block Drivers



We'll introduce block device drivers. We'll consider block buffering. We'll talk about what they are and how they are registered and unregistered. We'll discuss the important **gendisk** data structure. We'll discuss the block driver request function and see how reading and writing block devices is quite different than for character devices.

35.1 What are Block Drivers?

Drivers for **block devices** are similar in some ways to those for character drivers, but differences are many and deep.

In normal usage, block devices contain formatted and mountable filesystems, which allow random (non-sequential) access. The device driver does not depend on the type of filesystem put on the device.

While a particular system call may request any number of bytes, the low-level read/write requests must be in multiples of the block size. All access is **cached** (unless explicitly requested otherwise) which means writes to the device may be delayed, and reads may be satisfied from cache.

The drivers do not have their own read/write functions. Instead they deploy a **request function**, a callback function which is invoked by the higher levels of the kernel in a fluid way that depends on the use of the cache.

Block devices may have multiple **partitions**. In most instances the partition number corresponds to the device's minor number, while the whole device shares the same major number. The naming convention for the nodes is:

Major Name — Unit — Partition

e.g., `/dev/hdb4` has a Major Name of `hd`, is Unit `b` (the second), and is Partition 4. Details of the partitioning are contained in the `gendisk` data structure.

Block devices may also employ removable media such as CD-ROMS and floppy disks.

35.2 Buffering

Files reside on block devices which are organized in fixed size blocks, although I/O requests, made with system calls, may be for any number of bytes. Thus block devices must be controlled by a buffering/caching system, which is shared for all devices.

The blocks are cached through the page cache, and a given page may contain more than one block device buffer.

The device itself should only be accessed if:

- A block not presently in cache must be loaded on a read request.
- A block needs to be written (eventually) if the cache contents no longer match what is on the device itself. In this case the block must be marked as *dirty*. Note if a file is opened with the `O_SYNC` flag, no delay is allowed.

At periodic intervals the **pdflush** system process which causes all modified blocks that haven't been used for a certain amount of time, to be flushed back to the device. Other events may also trigger the flushing, with the object being to keep the number of ***dirty*** blocks in the cache at a minimum, and to make sure that the most important blocks, those describing inodes and superblocks, are kept most consistent.

The **sync** command writes all modified buffer blocks in the cache. The `fsync()` system call writes back all modified buffer blocks for a single file.

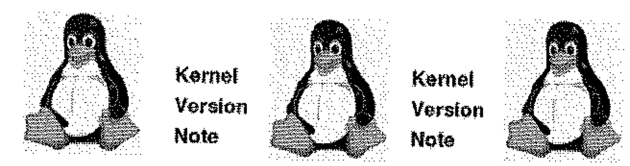
35.3 Registering a Block Driver

Registering a block device is generally done during the initialization routine, and in most ways is pretty similar to doing it for a character device. Unregistering is generally done during the cleanup routine, just as for a character device. The functions for doing this are:


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

int register_blkdev (unsigned int major, const char *name);
int unregister_blkdev (unsigned int major, const char *name);
```

`register_blkdev()` returns 0 on success and `-EBUSY` or `-EINVAL` on failure. Dynamic assignment is permitted. The value of `major` has to be less than or equal to `MAX_BLKDEV=255`.

`unregister_blkdev()` returns 0 on success and `-EINVAL` on failure. It checks that `major` is valid and that `name` matches with `major`, but doesn't check if you are the owner of the device you are unregistering.



- There also exist more modern block device registration and unregistration functions, `blk_register_region()` and `blk_unregister_region()`.
- The use of these is somewhat complicated and can be read about in an article by John Corbet in his driver porting series: **http://lwn.net/Articles/25711/**.

The `block_device_operations` structure plays the same role the `file_operations` structure plays for character drivers. It gets associated with the device through an entry in the `gendisk` data structure, as we will show shortly.

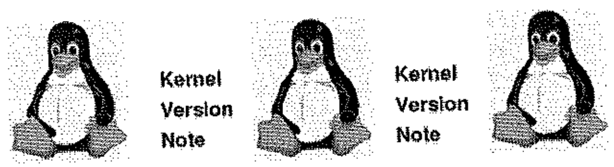
The `block_device_operations` structure is defined in **/usr/src/linux/include/linux/fs.h** as:

```
struct block_device_operations {
   int (*open) (struct inode * i, struct file *f);
   int (*release) (struct inode *i, struct file *f);
   int (*ioctl) (struct inode *i, struct file *f, unsigned cmd, unsigned long arg);
   long (*unlocked_ioctl) (struct file *f, unsigned cmd, unsigned long arg);
   long (*compat_ioctl) (struct file *f, unsigned cmd, unsigned long arg);
   int (*direct_access) (struct block_device *bdev, sector_t sector, void **kaddr,
                         unsigned long *pfn);
   int (*media_changed) (struct gendisk *gd);
   int (*revalidate_disk) (struct gendisk *gd);
   int (*getgeo)(struct block_device *bdev, struct hd_geometry *geo);
   struct module *owner;
};
```

For simple drivers, one need not even define `open()` and `release()` entry points, as generic ones will do the basic work. However, real hardware will probably need to perform certain steps at these times and will still need specific methods to be written.

Example:

```
static struct block_device_operations mybdrv_fops = {
    .owner=   THIS_MODULE,
    .open=    mybdrv_open,
    .release= mybdrv_release,
    .ioctl=   mybdrv_ioctl
};
```



- The 2.6.28 kernel introduces changes to the `block_device_operations` structure which is now defined as:

```
struct block_device_operations {
    int (*open) (struct block_device *bdev, fmode_t mode);
    int (*release) (struct gendisk *gd, fmode_t mode);
    int (*locked_ioctl) (struct block_device *bdev, fmode_t mode,
                         unsigned cmd, unsigned long arg);
    int (*ioctl) (struct block_device *bdev, fmode_t mode,
                  unsigned cmd, unsigned long arg);
    long (*compat_ioctl) (struct block_device *bdev, fmode_t mode,
                  unsigned cmd,  unsigned long arg);
    int (*direct_access) (struct block_device *bdev,
                          sector_t sector, void **kaddr,
                          unsigned long *pfn);
    int (*media_changed) (struct gendisk *gd);
    int (*revalidate_disk) (struct gendisk *gd);
    int (*getgeo)(struct block_device *bdev, struct hd_geometry *geo);
    struct module *owner;
};
```

35.4 gendisk Structure

The gendisk structure is defined in **/usr/src/linux/include/linux/genhd.h** and describes a paritionable device. You'll have to set it up, manipulate it, and free it when done.

The gendisk structure is:

```
struct gendisk {
    int major;                 /* major number of driver */
    int first_minor;
```



```
    int minors;                 /* maximum number of minors, =1 for
                                 * disks that can't be partitioned. */
    char disk_name[32];         /* name of major driver */
    struct hd_struct **part;  /* [indexed by minor] */
    int part_uevent_suppress;
    struct block_device_operations *fops;
    struct request_queue *queue;
    void *private_data;
    sector_t capacity;

    int flags;
    struct device *driverfs_dev;
    struct kobject kobj;
    struct kobject *holder_dir;
    struct kobject *slave_dir;

    struct timer_rand_state *random;
    int policy;

    atomic_t sync_io;       /* RAID */
    unsigned long stamp;
    int in_flight;
#ifdef   CONFIG_SMP
    struct disk_stats *dkstats;
#else
    struct disk_stats dkstats;
#endif
};
```

`major` is the major number associated with the device, and `first_minor` is the first minor number for the disk.

`disk_name` is the disk name without partition number; e.g., `hdb`.

`fops` points to the `block_device_operations` structure. Putting it in the `gendisk` structure is how it is associated with the device..

`request_queue` points to the queue of pending operations for the disk. Note there is only one request queue for the entire disk, not one for each partition.

`private data` points to an object not used by the kernel and thus can be used to hold a data structure for the device that the driver can use for any purpose.

`capacity` is the size of the disk in **512 byte** sectors; even if you have a different sector size, the capacity has to be unitized in this way.

`flags` control the way the device operates. Possible values include `GENHD_FL_REMOVABLE`, `GENHD_FL_CD` etc.

The following functions are used to allocate, configure, and free `gendisk` data structures:

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

struct gendisk *alloc_disk (int minors);
void add_disk (struct gendisk *disk);
void put_disk (struct gendisk *disk);
```

```
void del_gendisk (struct gendisk *disk);
void set_disk_ro (struct gendisk *disk);
```

The first step is to create the gendisk data structure. This is done with alloc_disk(), whose argument is the largest number of minor numbers, and thus partitions, the disk can accommodate.

One then fills in the various fields, such as the major number, the first minor (generally 0), and the capacity (which can be done with the void set_capacity (struct *gendisk, int nsectors) macro, and point to the proper request queue and device operations table.

Once any needed initializations are done to the device, the function add_disk() is called to activate the device. This increases the reference count for the disk; the function put_disk() should be called when the structure is released to decrement the reference count.

Upon removal of the device, one has to call del_gendisk(), although the actual removal won't happen until you subsequently call put_disk().

To put all partitions on the disk in a read-only status, you can use set_disk_ro().

35.5 Request Handling

Upper levels of the kernel handle the I/O requests associated with the device, and then group them in an efficient manner and place them on the **request queue** for the device, which causes them to get passed to the driver's **request function**.

The kernel maintains a request queue for each major number (by default). The data structure is of type struct request_queue and is defined in **/usr/src/linux/include/linux/blkdev.h**. The other major data structure involved is of type struct request and details each request being made to the driver. The request queue must be initialized and cleaned up with the functions:

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

struct request_queue *blk_init_queue (request_fn_proc *request, spinlock_t *lock);
void blk_cleanup_queue (struct request_queue *q);
```

and the sector size should be set in this structure with

```
void blk_queue_hardsect_size(struct request_queue *q, unsigned short size);
```

A spinlock has to be passed to the upper layers of the kernel. This will be taken out when the request function is called, with code like:

```
static spinlock_t lock;
....
spin_lock_init (&lock);
...
my_request_queue = blk_init_queue (my_request, &lock));
```

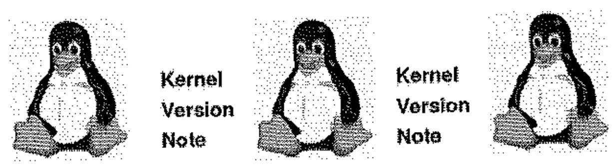
The simplest way to see how request handling is done is to look at a trivial request function:


```
static void my_request (struct request_queue *q){
    struct request *rq;
    int size;
    char *ptr;
    printk (KERN_INFO "entering request routine\n");

    while ((rq = elv_next_request (q))) {
        if (!blk_fs_request (rq)) {
            printk (KERN_INFO
            "This was not a normal fs request, skipping\n");
            end_request (rq, 0);
            continue;
        }
        ptr = my_dev + rq->sector * q->hardsect_size;
        size = rq->current_nr_sectors * q->hardsect_size;

        if ((ptr + size) > (my_dev + disk_size)) {
            printk (KERN_ERR " tried to go past end of device\n");
            end_request (rq, 0);
            continue;
        }
        if (rq_data_dir (rq)) {
            printk (KERN_INFO "a write\n");
            memcpy (ptr, rq->buffer, size);
        } else {
            printk (KERN_INFO "a read\n");
            memcpy (rq->buffer, ptr, size);
        }
        end_request (rq, 1);
    }
    printk ("KERN_INFO leaving request\n");
}
```

Peeling off the first request from the queue is done with elv_next_request() which returns NULL when there are no more requests. The function blk_fs_request() checks what kind of request is being delivered. This evaluates as true for normal filesystem requests, as opposed to diagnostic and other kinds of operations.

The actual copying is done with a simple memcpy(). Note, however, the use of the function rq_data_dir(), which checks the first bit of the flags field of the request structure which is set for writes, and cleared for reads.

Exiting the request function ends when the end_request() function is called with a second argument of 0 for failure, or 1 for success. (The other argument is a pointer to the request structure.)



The 2.6.31 kernel introduces changes in the block driver interface, reworking the `request_queue` structure and introducing some new functions. In addition the functions `elv_next_request()` and `end_request()` no longer appear. Furthermore the function `blk_queue_hardsect_size()` should be replaced with `blk_queue_logical_block_size()` which has the same arguments.
Here is an example of a reworked request function:

```
static void my_request (struct request_queue *q){
    struct request *rq;
    int size, res=0;
    char *ptr;
    unsigned nr_sectors, sector;
    printk (KERN_INFO "entering request routine\n");

    rq = blk_fetch_request(q);
    while (rq){
        if (!blk_fs_request (rq)) {
            printk (KERN_WARNING "This was not a normal fs request, skipping\n");
            goto done;
        }
        nr_sectors = blk_rq_cur_sectors(rq);
        sector = blk_rq_pos(rq);
        ptr = my_dev + sector * sector_size;
        size = nr_sectors * sector_size;

        if ((ptr + size) > (my_dev + disk_size)) {
            printk (KERN_WARNING " tried to go past end of device\n");
            goto done;
        }
        if (rq_data_dir (rq)) {
            printk (KERN_INFO "writing at sector %d, %ud sectors \n",
                    sector, nr_sectors);
            memcpy (ptr, rq->buffer, size);
        } else {
            printk (KERN_INFO "reading at sector %d, %ud sectors \n",
                    sector, nr_sectors);
            memcpy (rq->buffer, ptr, size);
        }
     done:
        if (!__blk_end_request_cur(rq, res))
            rq = blk_fetch_request(q);
     }
     printk (KERN_INFO "leaving request\n");
}
```


35.6 Labs

Lab 1: Building a Block Driver

Write a basic block device driver.

You'll need to implement at least the `open()` and `release()` entry points, and include a request function.

You can safely use 254 for the major device number and select a minor device number. For an added exercise try getting a major number dynamically. Assuming you are using **udev**, the node should be made automatically when you load the driver; otherwise you will have to actually add the node with the `mknod` command.

Keep track of the number of times the node is opened. Try permitting multiple opens, or exclusive use.

Write a program to read (and/or write) from the node, using the standard **Unix** I/O functions (`open()`, `read()`, `write()`, `close()`). After loading the module with `insmod` use this program to access the node.

NOTE: Make sure you have enough memory to handle the ram disk you create; The solution has 128 MB allocated.

Lab 2: Mountable Read/Write Block Driver

Extend the previous exercise in order to put an **ext3** file system (or another type) on your device.

You can place a filesystem on the device with

```
mkfs.ext3 /dev/mybdrv
mount /dev/mybdrv mnt
```

where you give the appropriate name of the device node and mount point.

For an additional enhancement, try partitioning the device with **fdisk**. For this you may need an additional `ioctl()` for `HDIO_GETGEO`, and you'll have to include: `linux/hdreg.h`. This **ioctl** returns a pointer to the following structure:

```
struct hd_geometry {
      unsigned char heads;
      unsigned char sectors;
      unsigned short cylinders;
      unsigned long start;
};
```

Remember the total capacity is (sector size) x (sectors/track) x (cylinders) x (heads). You also want to use a value of 4 for the starting sector.

If you are using a recent kernel and version of **udev**, the partition nodes should be made automatically when you load the driver; otherwise you will have to actually add them manually.

Index