Writing Linux Device Drivers

a guide with exercises

Including the 2.6.31 kernel

Jerry Cooperstein
# CONTENTS

## 4.1 Device Nodes ................................. 36
## 4.2 Major and Minor Numbers ................. 36
## 4.3 Reserving Major/Minor Numbers .......... 38
## 4.4 Accessing the Device Node ............... 40
## 4.5 Registering the Device ................... 41
## 4.6 udev and HAL ............................... 42
## 4.7 file_operations Structure ............... 44
## 4.8 Driver Entry Points ....................... 46
## 4.9 The file and inode Structures .......... 49
## 4.10 Module Usage Count ....................... 51
## 4.11 Labs ..................................... 52

## 5 Kernel Configuration and Compilation .......... 53
### 5.1 Installation and Layout of the Kernel Source .... 53
### 5.2 Kernel Browsers ............................ 56
### 5.3 Kernel Configuration Files ............... 56
### 5.4 Rolling Your Own Kernel .................. 57
### 5.5 initrd and initraads ..................... 60
### 5.6 Labs .................................. 63

## 6 Kernel Features ................................ 67
### 6.1 Components of the Kernel .................. 67
### 6.2 User-Space vs. Kernel-Space .............. 69
### 6.3 Scheduling Algorithms and Task Structures .... 70
### 6.4 Process Control ........................... 71
### 6.5 Labs .................................. 72

## 7 Kernel Style and General Considerations ........ 75
### 7.1 Coding Style .............................. 76
### 7.2 kernel-doc ................................ 77
### 7.3 Using Generic Kernel Routines and Methods ...... 77
### 7.4 Making a Kernel Patch ..................... 78
### 7.5 sparse .................................. 79
### 7.6 Using likely() and unlikely() ............... 80
### 7.7 Linked Lists ................................ 81
### 7.8 Writing Portable Code - 32/64-bit, Endianness .... 85
### 7.9 Writing for SMP ............................ 85
### 7.10 Writing for High Memory Systems ............ 86

## 8 Interrupts and Exceptions ...................... 89
### 8.1 What are Interrupts and Exceptions? ........ 90
### 8.2 Exceptions ............................... 90
### 8.3 Interrupts ................................. 92
### 8.4 MSI ...................................... 94
### 8.5 Enabling/Disabling Interrupts .............. 95
### 8.6 What You Cannot Do at Interrupt Time ......... 96
### 8.7 IRQ Data Structures ....................... 96
### 8.8 Installing an Interrupt Handler ............ 99
### 8.9 Labs .................................. 101

## 9 Modules II: Exporting, Licensing and Dynamic Loading .... 103
### 9.1 Exporting Symbols .......................... 104
### 9.2 Module Licensing ........................... 104
### 9.3 Automatic Loading/Unloading of Modules ........ 106
### 9.4 Built-in Drivers ........................... 107
### 9.5 Kernel Building and Makefiles .............. 109
### 9.6 Labs .................................. 110

## 10 Debugging Techniques .......................... 113
### 10.1 kmsg Messages ............................ 113
### 10.2 Kernel Debuggers .......................... 116
### 10.3 dbgifs .................................. 118
### 10.4 kprobes and jprobe ............ 119
### 10.5 Labs .................................. 122

## 11 Timing and Timers ............................. 125
### 11.1 Jiffies ................................. 125
### 11.2 Time Stamp Counter ....................... 127
### 11.3 Inserting Delays .......................... 128
### 11.4 What are Dynamic Timers? ................. 129
### 11.5 Timer Functions ........................... 129
### 11.6 Timer Implementation ..................... 130
### 11.7 High Resolution Timers ................... 131
19 Interrupt Handling and Deferrable Functions 225
  20.1 Top and Bottom Halves .............................. 225
  20.2 Deferrable Functions and softirqs .................. 227
  20.3 Tasklets ......................................... 228
  20.4 Work Queues ..................................... 231
  20.5 Creating Kernel Threads ......................... 234
  20.6 Threaded Interrupt Handlers ..................... 235
  20.7 Labs ............................................. 235

21 Hardware I/O 239
  21.1 Buses and Ports .................................. 240
  21.2 Memory Barriers .................................. 240
  21.3 Registering I/O Ports ......................... 241
  21.4 Resource Management ........................... 242
  21.5 Reading and Writing Data from I/O Registers ... 244
  21.6 Slowing I/O Calls to the Hardware ............. 245
  21.7 Allocating and Mapping I/O Memory ............ 246
  21.8 Accessing I/O Memory ........................... 247
  21.9 Access by User - iperf(), iopl(), /dev/port ... 248
  21.10 Labs ............................................ 249

22 PCI 253
  22.1 What is PCI? ..................................... 253
  22.2 PCI Device Drivers ............................ 256
  22.3 PCI Structures and Functions .................. 258
  22.4 Accessing Configuration Space ............... 259
  22.5 Accessing I/O and Memory Spaces ............ 260
  22.6 PCI Express ................................... 261
  22.7 Labs ............................................. 261

23 Direct Memory Access (DMA) 263
  23.1 What is DMA? .................................... 264
  23.2 DMA and Interrupts ............................. 264
  23.3 DMA Memory Constraints ....................... 265
  23.4 DMA Directly to User .......................... 266
  23.5 DMA under PCI .................................. 266
  23.6 DMA Pools ..................................... 269
  23.7 Scatter/Gather Mappings ....................... 269

24 Network Drivers I: Basics 273
  24.1 Network Layers and Data Encapsulation ......... 273
  24.2 DataLink Layer ................................ 276
  24.3 Network Device Drivers ....................... 276
  24.4 Loading/Unloading ............................. 277
  24.5 Opening and Closing ........................... 278
  24.6 Labs ............................................ 279

25 Network Drivers II: Data Structures 281
  25.1 net.device Structure ........................... 281
  25.2 net.device.ops Structure ...................... 287
  25.3 sk.buff Structure ................................ 289
  25.4 Socket Buffer Functions ....................... 290
  25.5 Labs ............................................. 293

26 Network Drivers III: Transmission and Reception 295
  26.1 Transmitting Data and Timeouts ............... 295
  26.2 Receiving Data ................................ 297
  26.3 Statistics .................................... 297
  26.4 Labs ............................................. 298

27 Network Drivers IV: Selected Topics 301
  27.1 Multicasting ................................... 302
  27.2 Changes in Link State .......................... 303
  27.3 ioctxs ......................................... 303
  27.4 NAPI and Interrupt Mitigation ............... 304
  27.5 NAPI Details .................................. 304
  27.6 TSO and TOE ................................... 305
  27.7 MII and ethtool ............................... 306

28 USB Drivers 309
  28.1 What is USB? .................................... 310
  28.2 USB Topology .................................. 310
  28.3 Descriptors .................................... 311
  28.4 USB Device Classes ........................... 312
  28.5 Data Transfer ................................ 313
  28.6 USB under Linux ............................... 314
List of Figures

2.1 From application to device ........................................ 13
2.2 USB: Controller, Core and Device ............................. 14
2.3 Using binary blobs in drivers .................................... 15
4.1 Accessing device nodes ............................................ 41
6.1 Main kernel tasks .................................................. 68
6.2 User and kernel space ............................................. 69
17.1 User and kernel address regions ............................... 184
17.2 DMA, normal and high memory ................................ 185
24.1 Network layers ................................................... 274
24.2 Data Packet Encapsulation ..................................... 275
25.1 Socket buffer layout .............................................. 289
28.1 USB topology .................................................... 311
28.2 USB descriptors .................................................. 311
28.3 USB: Controller, Core and Device ............................ 314
# List of Tables

2.3 `printk()` logging levels ................................................. 20

4.1 Device node macros ....................................................... 37
4.5 file structure elements ................................................... 49
4.6 inode structure elements ................................................ 50

5.1 Layout of the kernel source ................................................ 54

8.1 32-bit x86 exceptions ...................................................... 91
8.2 IRQ status values ........................................................ 97
8.3 IRQ handler flags ........................................................ 98
8.5 IRQ handler return values ................................................. 100

9.1 Licenses ........................................................................ 105

11.1 Timer groups ............................................................... 131

13.1 `ioctl()` return values .................................................... 154
13.3 `ioctl()` command bit fields ........................................... 157

17.1 GFP memory allocation flags ............................................ 187
17.3 Memory cache flags ....................................................... 191

18.2 `mmap()` memory protection bits ..................................... 202
18.3 `mmap()` flags .......................................................... 203

19.1 `poll()` flags ............................................................ 221

21.2 Serial mouse nodes and registers ..................................... 250

22.1 PCI features .................................................................. 255
22.2 pci.driver structure elements .......................................... 257
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 APIs 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 over 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 Unix or Linux 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 inside 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 five 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 wave propagation 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 had 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 is not just a repackaging of the course that was 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](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 Bisel 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 Koech, and Bill Kerr.

I would also like to thank Alessandro Rubinini 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 with 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 ............................................. 1
1.2 Linux Distributions .................................... 3
1.3 Kernel Versions ....................................... 4
1.4 Platforms ............................................. 6
1.5 Hardware ............................................. 7
1.6 Linux Driver Project ................................ 7
1.7 Documentation and Links ............................. 8

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 xzvf 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 SRDBS=a.22
  make KROOT=/lib/modules/2.6.31/build
  make SRDBS=a.0 b.23 KROOT=/usr/src/linux-2.6.31/build
  make ARCH=x86
```

where KROOT points to the kernel source files. On an x86.64 platform, specifying ARCH=x86 will compile 32-bit modules. The `gemmake` 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) 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.

---

### 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://fwn.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 those 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 exercises 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 latest stable version of the Linux kernel is:

2.6.38.5

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-git12

The latest 2.4 version of the Linux kernel is:

2.4.37.5

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>

...?

#define LINUX_VERSION_CODE < KERNEL_VERSION(2.6, 0)
\call_user_module_init(md, args, enp);
\also
\call_user_module_init(md, args, enp, wait);

...?

- Such explicit version dependence is frowned 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.
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/am/init.c we have arch/x86/am/init_32.c and arch/x86/am/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 licence.

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 | pdnacs | pdfdocs | rtfdocs }
```

the different forms giving you the documentation in either as web-browser, 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 than 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**


The full text of the book can be viewed or downloaded at http://www.kroah.com/ln/. (Disclaimer: The author was a technical reviewer on various editions of the previous four O’Reilly books.)


**Kernel Development and Mailing List Sites**

http://lwn.net

The 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://lwn.net/articles/how-participate-linux-community

A complete view of the kernel development process and how to join it.

1.7. DOCUMENTATION AND LINKS

http://lwn.net/Articles/driver-porting

A compendium of Jonathan Corbet’s articles on porting device drivers to the 2.6 kernel.


Tracks ongoing kernel developments that are likely to achieve incorporation in the near future.

http://lkm.org/

Archive of the Kernel Mailing List, updated in real time.

http://www.linux.org/lkm

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.


Andrew Morton’s guide to the perfect patch.

http://linux.yzz.us/patch-format.html

A detailed guide for how to submit patches to the official kernel tree.

http://linux.yzz.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://www.linux.com/coop/linux_pubs/

Several articles and talks by Jerry Cooperstein about changes for the 2.6 kernel.


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 ........................................ 11
2.2 Mechanism vs. Policy .................................... 14
2.3 Avoiding Binary Blobs .................................... 14
2.4 How Applications Use Device Drivers .................. 16
2.5 Walking Through a System Call ......................... 16
2.6 Error Numbers ............................................ 18
2.7 printk() .................................................. 19
2.8 Labs ....................................................... 21

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 pieces 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/lp

**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

**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 devices 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.
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 ifctFs, /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.

Well known examples of this method are employed by Nvidia graphics drivers, ndiswrapper wireless and NIC drivers, and various nfs 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 provocations.

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:

```c
    char *buf = malloc(nbytes);
    fd = open("/dev/udriver", 0_RDWR);
    nrread = read(fd, buf, nbytes);
```

(Of course we are giving only code fragments and being sloppy about error checking and all that!)
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/asn-generic/errno-base.h`:

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

Usually one returns -EBUSY, 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:

```c
#include <errno.h>
if ( ioctl(fd, ODB, arg) < 0 )
    perror("MY_DRIVER_IOCTL_CALL");
    return(errno);
```

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\", 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 KB_ERR_UBUS "<U>" /* system is unusable */
#define KB_ERR_URMT "<R>" /* action must be taken immediately */
#define KB_ERR_CRIT "<C>" /* critical conditions */
#define KB_ERR_ERR "<E>" /* error conditions */
#define KB_ERR_WARN "<W>" /* warning conditions */
#define KB_ERR_NOTICE "<N>" /* normal but significant condition */
#define KB_ERR_INFO "<I>" /* informational */
#define KB_ERR_DEBUG "<D>" /* debug-level messages */
```

The loglevel (or priority) forces an informational string to be prepended 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

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>console_loglevel</td>
<td>Messages with a higher priority than this will be printed to the console.</td>
</tr>
<tr>
<td>default_message_loglevel</td>
<td>Messages without an explicit priority will be printed with this priority.</td>
</tr>
<tr>
<td>minimum_console_level</td>
<td>Minimum (highest) value to which console_loglevel can be set.</td>
</tr>
<tr>
<td>default_console_loglevel</td>
<td>Default value for console_loglevel.</td>
</tr>
</tbody>
</table>

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 then you 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 tail /var/log/messages. You can also access the messages by clicking on System→Administration→System Log in your Desktop menu.

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

what you will get out will be:

```
Dec 18 15:51:64 p3 kernel: A message
Dec 18 15:51:64 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 off the screen.

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 07:54:43 localhost kernel: [ 714.557486] Byre: module unloaded
```

### 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 only the line:

```
obj-m += lab1_chdev.o
```

If you then type

```
make dep
```

```
make -C path to kernel source -M-SFW 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 daemon will 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:

    lsmod lsmod drivex.x
    rmmod lsmod drivex.x

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 nonexistentfile > /dev/mycdrv
    dd if=/dev/null of=/dev/mycdrv count=1
    dd if=/dev/mycdrv of=/dev/null 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? .......................... 23
3.2 A Trivial Example - Hello World .......... 24
3.3 Module Utilities .......................... 25
3.4 Passing Parameters ....................... 27
3.5 Compiling a Module ...................... 28
3.6 Modules and Hot Plug .................... 33
3.7 Labs .......................... 34

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.
CHAPTER 3. MODULES I: BASICS

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.

```c
#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 GINNIS");
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. Labelling 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.

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 method 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

rmmmod

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

    rmmmod 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

3.4. PASSING PARAMETERS

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 modutils package requires use of the .ko extension with insmod, but modprobe and rmmmod 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, perms);
module_param_named (name, value, type, perms);
module_param_array (name, type, num, perms);
module_param_string (name, string, len, perms);

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

The perms 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
CHAPTER 3. MODULES I: BASICS

S_IRUGO (0644) 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_string() function is used to pass 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 __module__ macros) to pass parameters to the kernel on the boot command line. This makes it 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:

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

A list of all known possible 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:

```c
MODULE_AUTHOR();
MODULE_DESCRIPTION(desc);
MODULE_SUPPORTED_DEVICES(name);
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:

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

If you then type

```
make -C /lib/modules/$(uname -r)/build M=2.6 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=2.6 modules_install
```

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

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

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

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

```bash
#!/bin/bash
# Automatic kernel makefile generation
# Jerry Cooperstein, cooper@cooper.com, 2/2003 - 1/2009
# License: GPL v2

OBJS="*" # list of kernel modules (.o files)
KELLS="*" # list of kernel modules (.c files)
UKELLS="*" # list of userland programs (.c files)
UKELS="*" # list of userland programs (executables)
```

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

If you then type

```
make -C /lib/modules/$(uname -r)/build M=2.6 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=2.6 modules_install
```

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

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

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

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

```bash
#!/bin/bash
# Automatic kernel makefile generation
# Jerry Cooperstein, cooper@cooper.com, 2/2003 - 1/2009
# License: GPL v2

OBJS="*" # list of kernel modules (.o files)
KELLS="*" # list of kernel modules (.c files)
UKELLS="*" # list of userland programs (.c files)
UKELS="*" # list of userland programs (executables)
```
3.5. Compiling a Module

# skip empty directories
if [ -z "$find . -maxdepth 1 -name "*.c" -o -d "$find . -maxdepth 1 -name "*.c" -o -d "KNOOUT" ] ; then
  echo No need to make Makefile: no source code exists
  fi
  for names in *.c ; do
    # exclude files with NOMEM or .mod.c files
    if [ "$(grep \NOMEM \$names)" ] || [ "$(grep vermagic \$names)" ] ; then
      echo \$names is being skipped, it is not a module or program
    fi
    if [ "$(grep \<linux\>/module.h > \$names)" ] ; then
      FILENAME_DOTO=\$(basename \$names .c).o
      OBJ=\$\$LIBS \$FILENAME_DOTO
      K_B=\$\$CP \$names
    else
      if [ -f \$KNOOUT ] && [ -d \$KNOOUT ] ; then
        echo kernel source directory \$KNOOUT does not exist or has no Makefile
        exit 1
      fi
      # additional flags?
      if [ "\$KERNELSRC" != "" ] ; then
        CFLAGS_U.X="-D$CFLAGS_U.X -DKNOOUT/include" \n        CFLAGS_T.X="-DCFLAGS_T.X -DKNOOUT/include"
      fi
      # extract the VERSION info from the Makefile
      KV=$(grep "\VERSION = \$\$MVF | awk '{ print \$3;}'")
      KP=$(grep "\PATCHLEVEL = \$\$MVF | awk '{ print \$3;}'")
      KS=$(grep "\SUBLEVEL = \$\$MVF | awk '{ print \$3;}'")
      KF=$(grep "\EXTRAVERSION = \$\$MVF | awk '{ print \$3;}'")
      KERNEL=\$\$MVF.\$\$MVF.
      echo \$KERNEL, KV=\$KV KP=\$KP KS=\$KS KF=\$KF
      # check on the major release version
      if [ \$KV != 6 ] ; then
        echo KNOOUT=\$KNOOUT is not 2.6, exiting
        exit 1
      fi
      # construct lists of kernel and user sources and targets
      # make all targets
      if [ "\$U.X != "" ] ; then
        ALL=\$ALL ustrip=1 ; fi
      if [ "\$T.X != "" ] ; then
        ALL=\$ALL threadstrip=1 ; fi
      if [ "\$SLOTS != "" ] ; then
        ALL=\$ALL modules=1 ; fi
    # echo if you are curious
      echo K.S=\$\$S U.BJ=\$\$BJS U.S=\$\$U.S U.X=\$\$U.X T.S=\$\$T.S T.X=\$\$T.X
      echo "We're done preparing, let's build the makefile finally!"
      # get rid of the old Makefile, build a new one
    fi
  fi
fi
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:

```c
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:

```c
class const struct pci_device_id *skge_id_table[] = {
    { PCI_DEVICE(PCI_VENDOR_ID_SCOM, PCI_DEVICE_ID_SCOM_0C940),
    { PCI_DEVICE(PCI_VENDOR_ID_SCOM, PCI_DEVICE_ID_SCOM_0C9400),
    ...,
    { PCI_VENDOR_ID_LIKESYS, 0x0102, PCI_ANY_ID, 0x0016, },
    { 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/moddevicetable.h` and include:

```c
usb
pci
lsmod
input
eisa
ppp
srio
```

and each one has a structure associated with it, such as struct usb_device_id, struct ioeo1394_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.pcmmap, modules.unmapmap, etc.

When a new device is added to the system, those 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 .................................. 36
4.2 Major and Minor Numbers .................. 36
4.3 Reserving Major/Minor Numbers .......... 38
4.4 Accessing the Device Node ................. 40
4.5 Registering the Device ..................... 41
4.6 udev and HAL .................................. 42
4.7 file_operations Structure ................. 44
4.8 Driver Entry Points .......................... 46
4.9 The file and Inode Structures ............. 49
4.10 Module Usage Count ........................ 51
4.11 Labs ......................................... 52
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 26 01:56 cdrom -> hda
brw-rw---- 1 coop floppy 2, 0 May 26 01:56 fdd
brw-rw---- 1 coop floppy 2, 4 May 26 01:56 fdd0360
brw-rw---- 1 coop floppy 2, 16 May 26 01:56 fdd0720
crw-rw---- 1 root lp 99, 0 May 26 01:56 parport0
crw-rw---- 1 root lp 99, 1 May 26 01:56 parport1
crw-rw---- 1 root lp 99, 2 May 26 01:56 parport2
```

Device nodes are made with:

```
mkdir [-n node] /dev/name <type> <major> <minor>
e.g., mkdir -m 666 /dev/wc0trv c 254 1
```

or from the `mkdir()` 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>
<thead>
<tr>
<th>Macro</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAJOR(dev_t dev);</td>
<td>Extract the major number.</td>
</tr>
<tr>
<td>MINOR(dev_t dev);</td>
<td>Extract the minor number.</td>
</tr>
<tr>
<td>MKDEV(int major, int minor);</td>
<td>Return a dev_t built from major and minor numbers</td>
</tr>
</tbody>
</table>

One can also use the inline convenience functions:

```
unsigned minor(struct inode *inode); /* = MINOR(inode->rrdev) */
unsigned major(struct inode *inode); /* = MAJOR(inode->rrdev) */
```

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:

```c
$ stat vmlinuz-2.6.26-rc5
```

File: `vmlinuz-2.6.26-rc5'
Size: 2549888 Blocks: 8592 3D Blocks: 1024 regular file
Device: 6002/20534 Inode: 200900 Links: 1
Access: (0/44/-rw-x--x--) Uid: ( 0/ root) Gid: ( 0/ root)

shows the file resides on the disk partition with major number 8 and minor number 5 (`/dev/uda5'), which is listed at 805h (hexadecimal) or 20534 (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:

```c
#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 device 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 `mkdev` 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:

```c
#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 will 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:

  ```c
  #include <linux/fs.h>
  
  int register_chrdev (unsigned int major, const char *name, 
                        struct file_operations *ops);
  
  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 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:

```c
#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 modprobe 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:

```c
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 poll fd, 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 1/O library functions: fopen(), fclose(), fread(), fwrite(), and fseek().

### 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`:

```c
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:

```c
#include <linux/cdev.h>
```
4.6 UDEV AND HAL

UDEV handles the dynamical generation of device nodes but 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 ishal 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 simplest 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 using 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:

```c
#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 *name, ...);
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:

```c
static struct class *foo_class;

/* create node in the init function */
foo_class = class_create (TRUE_MODULE, "my_class");
device_create (foo_class, NULL, first, "Mkdev", "mydev", 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/fi.h, and looks like:

```
2.6.31:1486 struct file_operations {
2.6.31:1487  struct module *owner;
2.6.31:1488  loff_t (*close) (struct file *, loff_t, int);
2.6.31:1489  sise_t (*read) (struct file *, char __user *, size_t, loff_t t);
2.6.31:1490  sise_t (*write) (struct file *, char __user *, size_t, loff_t t);
2.6.31:1491  sise_t (*ioRead) (struct klock *, const struct lovec *, unsigned long, loff_t t);
2.6.31:1492  sise_t (*io_write) (struct klock *, const struct lovec *, unsigned long, loff_t t);
2.6.31:1493  int (*fdir) (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 *, struct file_operations *, unsigned int, unsigned long);
2.6.31:1497  long (*compat_ioctl) (struct file *, struct file_operations *, unsigned int, unsigned long);
2.6.31:1498  int (*unmp) (struct file *, struct mm_area struct *);
2.6.31:1499  int (*open) (struct inode *, struct file *);
2.6.31:1500  int (*flush) (struct file *, fl_owner_t *);
2.6.31:1501  int (*release) (struct inode *, struct file *);
2.6.31:1502  int (*fsync) (struct file *, struct dentry *, int dataasync);
2.6.31:1503  int (*fasync) (struct klock *, int, int, int);
2.6.31:1504  int (*lock) (struct file *, int, struct file_lock *);
2.6.31:1505  sise_t (*unmappage) (struct file *, struct page *, int, size_t, loff_t t, int);
2.6.31:1506  unsigned long (*get_unmapped_area) (struct file *, unsigned long, long, unsigned long, unsigned long, unsigned long);
2.6.31:1507  int (*check_flags) (int);
2.6.31:1508  sise_t (*lock) (struct file *, struct file_lock *);
2.6.31:1509  sise_t (*unmap_pages) (struct file *, loff_t t, struct file_operations *, size_t, unsigned int);
2.6.31:1510  sise_t (*splice_write) (struct pipe_inode_info *, struct file *, loff_t t, size_t, unsigned int);
2.6.31:1511  sise_t (*splice_read) (struct file *, loff_t t, struct pipe_inode_info *, size_t, unsigned int);
2.6.31:1512  int (*ioctl) (struct file *, long, struct file_lock *);
2.6.31:1513  struct inode_operations {
2.6.31:1514  int (*create) (struct inode *, struct dentry *, int, struct userdata *);
2.6.31:1515  int (*lookup) (struct inode *, struct dentry *, struct userdata *);
2.6.31:1516  int (*link) (struct dentry *, struct inode *, struct dentry *);
2.6.31:1517  int (*unlink) (struct inode *, struct dentry *);
2.6.31:1518  int (*rmdir) (struct inode *, struct dentry *, struct dentry *);
2.6.31:1519  int (*rmdir) (struct inode *, struct dentry *, struct dentry *);
2.6.31:1520  int (*rename) (struct inode *, struct dentry *, struct dentry *);
2.6.31:1521  int (*rename) (struct inode *, struct dentry *, struct dentry *);
2.6.31:1522  int (*mknod) (struct inode *, struct dentry *, int dev_t);
2.6.31:1523  int (*chmod) (struct inode *, struct dentry *, struct dentry *);
2.6.31:1524  int (*chmod) (struct inode *, struct dentry *, struct dentry *);
2.6.31:1525  int (*chmod) (struct inode *, struct dentry *, struct dentry *);
2.6.31:1526  int (*chmod) (struct inode *, struct dentry *, struct dentry *);
2.6.31:1527  int (*chmod) (struct inode *, struct dentry *, struct dentry *);
2.6.31:1528  int (*chmod) (struct inode *, struct dentry *, struct dentry *);
```

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 area. The name is true with the file and inode structures to be
discussed shortly.

The file operations structure is initialized with code like:

```
struct file_operations form = {
    .owner = THIS MIDDLE,
    .open = my_open,
    .release = my_close,
    .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.

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 mode 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 map().

  - A generic default method will be invoked. An example is lseek(). Sometimes
    this means the method will always succeed. Examples are open() and release().

- 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 map().

  - A generic default method will be invoked. An example is lseek(). Sometimes
    this means the method will always succeed. Examples are open() and release().

- 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 map().

  - A generic default method will be invoked. An example is lseek(). Sometimes
    this means the method will always succeed. Examples are open() and release().
4.8 Driver Entry Points

```c
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 (*fseek) (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 mask the FMODE_SEEK bit in the file file structure (probably during the open() method) as in:

```c
    file->f_mode = file->f_mode & ~FMODE_SEEK;
```  

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

read 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:

```c
static size_t mydev_read (struct file *filp, char __user *buf, size_t size, loff_t *ppos)
{
    int nbytes = -1;
    char *bufn = copy_to_user (buf, blkdev->ppos, blkdev);  
    *ppos += nbytes;
    printk(KERN_INFO "\a READING function, blkdev=%d, pos=0x%p", nbytes, (int) *ppos);
    return nbytes;
}
```

In this simple case a read merely copies from a buffer in kernel-space (bufn) to a buffer in user-space (buf). But one can not use copyto() 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:

```c
#include <linux/slab.h>
char *bufn = kmalloc (blkdev->size, GFP_KERNEL);
....
free (bufn);
```

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

4.8. DRIVER ENTRY POINTS

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.

```c
size_t (*write) (struct file *filp, const char __user *buf, 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:

```c
int nbytes = blkdev->copy_from_user (bufn, *ppos, blkdev);  
```

for the same reason

```c
int (*readdir) (struct file *filp, void *data, 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.

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

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.

```c
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.

```c
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.

```c
int (*remap) (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.

```c
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.
CHAPTER 4. CHARACTER DEVICES

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 fdatasync);

is used to flush any pending data.

int (*fsync) (int, struct file *filp, int);

checks the device 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 nio
v, loff_t pos);

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

int (*aio_fsync) (struct kiocb *, int fdatasync);

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_a
ctor_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().

4.9. THE FILE AND INODE STRUCTURES

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct path</td>
<td>Gives information about the file directory entry, including a pointer to the inode.</td>
</tr>
<tr>
<td>f_path</td>
<td>Operations associated with the file. Can be changed when the method is invoked again.</td>
</tr>
<tr>
<td>const struct file_operations f_op</td>
<td>Identified by the bits LINUX_READ and LINUX_WRITE. Note the kernel checks permissions before invoking the driver.</td>
</tr>
<tr>
<td>loff_t f_pos</td>
<td>Current position in the file; a 64-bit value. While the final argument to the read() and write() entry points usually point to this, the lseek() 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.)</td>
</tr>
<tr>
<td>unsigned int f_flags</td>
<td>0_RDONLY, 0_RSYNCLOCK, 0_SYNC, etc. Needs to be checked for non-blocking operations.</td>
</tr>
</tbody>
</table>
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 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

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev_t i_rdev</td>
<td>Contains the actual device number from the major and minor numbers can be extracted.</td>
</tr>
<tr>
<td>dev_t i_cdev</td>
<td>Points back to the basic character driver structure.</td>
</tr>
</tbody>
</table>

---

### 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:

```c
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:

```c
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

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

and would usually be invoked as something like:

```c
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 .................................. 53
5.2 Kernel Browse .............................................................................. 56
5.3 Kernel Configuration Files ............................................................ 56
5.4 Rolling Your Own Kernel ............................................................... 57
5.5 initrd and initrams ....................................................................... 60
5.6 Labs ............................................................................................. 63

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.)
CHAPTER 5. KERNEL CONFIGURATION AND COMPILATION

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 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

<table>
<thead>
<tr>
<th>Directory</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>arch</td>
<td>1386, i864, alpha, arm, sparc, sparc64, mips, mips64, a68k, ppc, sparc ... Architecture specific code for boot, synchronization, memory and process management.</td>
</tr>
<tr>
<td>kernel</td>
<td>Generic main kernel routines.</td>
</tr>
<tr>
<td>mm</td>
<td>Generic memory management code. swapping, mmapping, kernel malloc, etc.</td>
</tr>
<tr>
<td>init</td>
<td>Generic kernel start-up code.</td>
</tr>
<tr>
<td>drivers</td>
<td>char, block, net, scsi, fs, cdlrom, pci ... Device drivers sorted by type.</td>
</tr>
<tr>
<td>sound</td>
<td>ALSA (Advanced Linux Sound Architecture), including sound card drivers.</td>
</tr>
<tr>
<td>block</td>
<td>Low-level infrastructure for the block device layer. Specific block device drivers are under drivers/block.</td>
</tr>
<tr>
<td>fs</td>
<td>Filesystems, with subdirectories for each type.</td>
</tr>
<tr>
<td>net</td>
<td>Ethernet, ip, deernet, ipx, ipv4, ipv6, appletalk and other network code.</td>
</tr>
<tr>
<td>security</td>
<td>Security models, including SELinux.</td>
</tr>
<tr>
<td>crypto</td>
<td>Cryptographic algorithms.</td>
</tr>
<tr>
<td>lib</td>
<td>Some standard library routines, mostly for strings.</td>
</tr>
<tr>
<td>ipc</td>
<td>System V Inter-Process Communications code.</td>
</tr>
<tr>
<td>uucp</td>
<td>User-space interaction; so far only intranets code.</td>
</tr>
<tr>
<td>scripts</td>
<td>Various scripts used to compile and package kernels.</td>
</tr>
<tr>
<td>Documentation</td>
<td>Various documentation resources; sometimes not up to date.</td>
</tr>
<tr>
<td>virt</td>
<td>Virtualization infrastructure.</td>
</tr>
<tr>
<td>firmware</td>
<td>Firmware that is packaged with kernel.</td>
</tr>
<tr>
<td>samples</td>
<td>Sample kernel code used for tracing, profiling and debugging purposes.</td>
</tr>
<tr>
<td>tools</td>
<td>User-space tools, used for performance counting.</td>
</tr>
<tr>
<td>include</td>
<td>System header files.</td>
</tr>
</tbody>
</table>

5.1. INSTALLATION AND LAYOUT OF THE KERNEL SOURCE

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

<table>
<thead>
<tr>
<th>Directory</th>
<th>.o files</th>
<th>.h files</th>
<th>.c files</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>310430</td>
<td>199276</td>
<td>846196</td>
<td>1076122</td>
</tr>
<tr>
<td>.drivers</td>
<td>1776</td>
<td>901738</td>
<td>5014669</td>
<td>5910883</td>
</tr>
<tr>
<td>.arch</td>
<td>305694</td>
<td>654436</td>
<td>1197999</td>
<td>2163133</td>
</tr>
<tr>
<td>.fs</td>
<td>0</td>
<td>72900</td>
<td>821466</td>
<td>894266</td>
</tr>
<tr>
<td>.net</td>
<td>0</td>
<td>15667</td>
<td>568563</td>
<td>578530</td>
</tr>
<tr>
<td>.sound</td>
<td>218</td>
<td>39800</td>
<td>464624</td>
<td>522842</td>
</tr>
<tr>
<td>.include</td>
<td>0</td>
<td>385396</td>
<td>0</td>
<td>385396</td>
</tr>
<tr>
<td>.kernel</td>
<td>0</td>
<td>2773</td>
<td>140979</td>
<td>143732</td>
</tr>
<tr>
<td>.usr</td>
<td>0</td>
<td>266</td>
<td>62289</td>
<td>62555</td>
</tr>
<tr>
<td>.security</td>
<td>0</td>
<td>4721</td>
<td>41171</td>
<td>45992</td>
</tr>
<tr>
<td>.crypto</td>
<td>0</td>
<td>9614</td>
<td>33396</td>
<td>42210</td>
</tr>
<tr>
<td>.lib</td>
<td>0</td>
<td>817</td>
<td>31856</td>
<td>32673</td>
</tr>
<tr>
<td>.scripts</td>
<td>0</td>
<td>2838</td>
<td>22263</td>
<td>25101</td>
</tr>
<tr>
<td>.block</td>
<td>0</td>
<td>167</td>
<td>19466</td>
<td>10372</td>
</tr>
<tr>
<td>.tools</td>
<td>0</td>
<td>1565</td>
<td>13772</td>
<td>16337</td>
</tr>
<tr>
<td>.Documentation</td>
<td>0</td>
<td>0</td>
<td>9155</td>
<td>9155</td>
</tr>
<tr>
<td>.sysfs</td>
<td>0</td>
<td>177</td>
<td>7320</td>
<td>7497</td>
</tr>
<tr>
<td>.virt</td>
<td>0</td>
<td>164</td>
<td>4103</td>
<td>4267</td>
</tr>
<tr>
<td>.init</td>
<td>0</td>
<td>76</td>
<td>2994</td>
<td>3070</td>
</tr>
<tr>
<td>.firmware</td>
<td>2425</td>
<td>0</td>
<td>268</td>
<td>2693</td>
</tr>
<tr>
<td>.samples</td>
<td>0</td>
<td>150</td>
<td>1032</td>
<td>1182</td>
</tr>
<tr>
<td>.usr</td>
<td>117</td>
<td>0</td>
<td>692</td>
<td>709</td>
</tr>
</tbody>
</table>

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 observe 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:

<table>
<thead>
<tr>
<th>Kernel Version</th>
<th>TOTAL LINES</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.18</td>
<td>7082962</td>
</tr>
<tr>
<td>2.6.19</td>
<td>7308043</td>
</tr>
<tr>
<td>2.6.20</td>
<td>7408043</td>
</tr>
<tr>
<td>2.6.21</td>
<td>7522286</td>
</tr>
<tr>
<td>2.6.22</td>
<td>7744727</td>
</tr>
<tr>
<td>2.6.23</td>
<td>7818168</td>
</tr>
<tr>
<td>2.6.24</td>
<td>8053258</td>
</tr>
<tr>
<td>2.6.25</td>
<td>8396801</td>
</tr>
<tr>
<td>2.6.26</td>
<td>8636933</td>
</tr>
<tr>
<td>2.6.27</td>
<td>8990888</td>
</tr>
<tr>
<td>2.6.28</td>
<td>9128690</td>
</tr>
<tr>
<td>2.6.29</td>
<td>9371260</td>
</tr>
<tr>
<td>2.6.30</td>
<td>10419687</td>
</tr>
<tr>
<td>2.6.31</td>
<td>1076122</td>
</tr>
</tbody>
</table>
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](http://lxr.linux.no). This website contains browsable 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/~qiyoung/lxr/source](http://users.sosdg.org/~qiyoung/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 `escscope` utility, which is a standard offering on most Linux systems. To use on the kernel sources one need merely run `make escscope` in the kernel source directory, creating the various index files needed, and then simply run `escscope` 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](http://www.gnu.org/software/global). Many distributions offer this as a package. Once `global` is installed one need only add 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/KERNEL/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](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 source 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 Linux 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_PROG` option is turned on, information can be read out directly from `/proc/config.gz`.

5.4 Rolling Your Own Kernel

Running the `scripts/extrac-tikconfig` 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](http://www.kernel.org). In order 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](http://www.kernel.org), and the second can be obtained from [http://www.cooj.com/LDD](http://www.cooj.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

```bash
$ 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.

```bash
#!/bin/bash

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

function get_SOURCE()
{
   KERNEL=$1
   TARFILE=$2
   CONFIGFILE=$3
   LRVP=linux-$(KERNEL)
   KOCONFIG=$(LRVP)/.config

   if [ -f "$STARFILE" ]; then
      echo "$STARFILE, aborting" | exit
   else
      echo "$CONFIGFILE, aborting" | exit
   fi

   # get SOURCE
   # get Sealable Source
   # get Compilation
   # get Installation

   # get Sealable Source
   # get Compilation
   # get Installation
}

get_SOURCE
```
function makeinitrd_REHAT();
# Construct mkinitrd image
# REHAT 3.2.6-16 kernels mocked this up
if [ "$(grep CONFIG_MD_RAID945 ./config)" = "" ];
    exit
else
    mkinitrd -v -f $BOOT/initrd-$KERNEL.img $KERNEL
fi

# Update /boot/grub/grub.conf
cp /boot/grub/grub.conf /boot/grub/grub.conf.OLD
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'sable 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_UBSOUT()
makeinitrd_DEBIAN
}

function makeinitrd_SUSE()
makeinitrd_DEBIAN
}

function makeinitrd_GENTOO()
make install
gendkernel ramdisk --kerneldir=$PWD
}

function makeinitrd_X() {
    echo System $SYSTEM is not something I understand, can't finish exit
}

# Start of the work
HANDS="$@
[ "$HANDS" = "$" ] && get_source $1 $2 $3
5.5. initrd and initramfs

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 bash 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 ramdisk 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 /etc/fstab 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:

$cpio -iv --from=./initrd-2.6.31.img |
$gzip -d
$ ls -lh

which gives:

total 32
-dwx------ 2 coop coop 4096 Jun 29 08:17 bin
-dwrxr-xr-x 5 coop coop 4096 Jun 29 08:17 dev
-dw-------- 2 coop coop 4096 Jun 29 08:17 mdev
-rwx-------- 1 coop coop 1579 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 ebins -> bin
-drwx------- 2 coop coop 4096 Jun 29 08:17 sys
-drwx------- 5 coop coop 4096 Jun 29 08:17 syslog

./bin:

total 3842
-rwx------ 1 coop coop 52528 Jun 29 08:17 insmod
-rwx-------------- 1 coop coop 19929 Jun 29 08:17 modprobe -> /sbin/nstab
-rwx------ 1 coop coop 2380982 Jun 29 08:17 nstabs

./dev:

total 4
-dwx------ 2 coop coop 4096 Jun 29 08:17 mapper
lrwxrwxrwx 1 coop coop 4 Jun 29 08:17 ram -> ram

./dev/mapper:

total 0

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 journaling 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.
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 config`, 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](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”

### 5.6. LABS

- **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](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.


In order to compile modules against your kernel source you need more than just a proper `.config` file. Short of running a compilation again, 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 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:
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 ........................................ 67
6.2 User-Space vs. Kernel-Space .................................... 69
6.3 Scheduling Algorithms and Task Structures .................. 70
6.4 Process Context ................................................... 71
6.5 Labs ............................................................... 72

6.1 Components of the Kernel

Process Management

- Creating and destroying processes.
- Input and output to processes.
6.2. USER-SPACE VS. KERNEL-SPACE

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.

Execution is transferred from user mode (space) to kernel mode (space) through system calls (which are implemented using asynchronous 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.

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.
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 preemtable 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->rgid` 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:

```c
struct task_struct *find_task_by_pid(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:

```c
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_t`’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 toiosely 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 (Completely Fair Scheduler) 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.
CHAPTER 6. KERNEL FEATURES

- 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:

```c
in_irq();     /* in hardware interrupt context */
in_softirq(); /* in software interrupt context */
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:

```bash
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

```bash
strace ls -lRF /dev | less
```
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 .................................................. 76
7.2 kernel-doc ...................................................... 77
7.3 Using Generic Kernel Routines and Methods ............... 77
7.4 Making a Kernel Patch .......................................... 78
7.5 sparse .......................................................... 79
7.6 Using likely() and unlikely() ................................. 80
7.7 Linked Lists .................................................... 81
7.8 Writing Portable Code - 32/64-bit, Endianness ............. 85
7.9 Writing for SMP ................................................ 85
7.10 Writing for High Memory Systems ......................... 86
7.11 Keeping Security in Mind .................................... 86
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 -nopro -tab -ts -space -crlf -as -cc1 "".

- 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 obfuscation. 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 of 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:

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


7.2 KERNAL-Doc

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-mano-HOWTO.txt suffices to demonstrate the format that must be used:

```c
/**
 * foo(x) - short function description of foo(x)
 * @args: Describe the first argument to foo(x).
 * @args2: Describe the second argument to foo(x).
 * @args3: One can provide multiple line descriptions
 *        for arguments.
 * @returns: A longer description, with more discussion of the function foo(x)
 *          that might be useful to those using or modifying it. Begins with
 *          empty comment line, and may include additional embedded empty
 *          comment lines.
 * @returns: 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 sent 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 [-y] ...

[-dbook | -html | -text | -man]
[-function functionname | -function functionname ...]
[-nfunction functionname | -nfunction functionname ...]
< 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:
7.5. SPARSE

- Your modified source should be cleaned up before producing the patch, perhaps using make arproper. 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 sources 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

cp 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.)


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 Scanline or Semantic 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 an 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:

```c
static ssize_t mycdv_read(struct file *file, char *buf, size_t count, loff_t *ppos);
... remove_proc_entry("ps", 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 mydisk_read(struct file *file, char __user *buf, size_t count, loff_t *ppos);

remove_proc_entry("proc", 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/](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.cademoukey.org.uk/projects/git-snapshots/sparse/](http://www.cademoukey.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.
CHAPTER 7. KERNEL STYLE AND GENERAL CONSIDERATIONS

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 name 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

```c
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:

```c
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 initialized prior to use. This can be done statically at compile time as:

```c
LIST_HEAD(my_list);
```

or

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

or at run time as:

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

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

```c
#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);
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:

```c
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 following code fragments accomplish the same thing:
LIST_HEAD (b);
struct list_head *a;
struct my_s *s;
list_for_each (l, &a) {
    s = list_entry (1, struct my_s, list);
    ...
}

or

LIST_HEAD (b);
struct my_s *s;
list_for_each_entry (a, &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_str", 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 (1, struct my_entry, list);
        fooobar (me->intvar);
        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 off_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 aboutendianness (big-endian vs. little-endian) should be avoided. Use the available kernel functions which are conscious ofendianness, 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

```c
#include <stdio.h>
#include <sys/types.h>
#include <errno.h>
```

7.13 Labs

7.13.1 Labs

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 smm 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 system 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:
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 cannot 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? .......................... 90
8.2 Exceptions ............................................. 90
8.3 Interrupts .............................................. 92
8.4 MSI ..................................................... 94
8.5 Enabling/Disabling Interrupts ............................... 95
8.6 What You Cannot Do at Interrupt Time ................... 96
8.7 IRQ Data Structures .................................... 96
8.8 Installing an Interrupt Handler ............................. 99
8.9 Labs ..................................................... 101
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 (Advanced Programmable Interrupt Controller.)

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-concurrent 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>
<thead>
<tr>
<th>#</th>
<th>Exception/Service Routine</th>
<th>Type</th>
<th>Signal</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Divide Error divide_error()</td>
<td>fault</td>
<td>SIGFPE</td>
<td>Attempted division by 0.</td>
</tr>
<tr>
<td>1</td>
<td>Debug debug_error()</td>
<td>trap or fault</td>
<td>SIGTRAP</td>
<td>Used by debugging and tracing programs.</td>
</tr>
<tr>
<td>2</td>
<td>NMI nmi()</td>
<td>None</td>
<td>None</td>
<td>Reserved for nonmaskable interrupts that use the NMI pins.</td>
</tr>
<tr>
<td>3</td>
<td>Breakpoint int3()</td>
<td>trap</td>
<td>SIGTRAP</td>
<td>Debugger has inserted a int3 instruction (breakpoint).</td>
</tr>
<tr>
<td>4</td>
<td>Overflow overflow()</td>
<td>trap</td>
<td>SIGSEGV</td>
<td>An into instruction has been executed and overflow detected.</td>
</tr>
<tr>
<td>5</td>
<td>Bounds Check bounds()</td>
<td>fault</td>
<td>SIGSEGV</td>
<td>A bounds instruction has been executed and the operand is outside of valid bounds.</td>
</tr>
<tr>
<td>6</td>
<td>Invalid opcode invalid_op()</td>
<td>fault</td>
<td>SIGILL</td>
<td>Bad opcode (part of the CPU instruction that selects the operation.)</td>
</tr>
<tr>
<td>7</td>
<td>Device not available device_not_available()</td>
<td>fault</td>
<td>SIGSEGV</td>
<td>A floating point or MMX instruction executed with the TS flag of cr0 set.</td>
</tr>
<tr>
<td>8</td>
<td>Double fault double_fault()</td>
<td>abort</td>
<td>SIGSEGV</td>
<td>An exception detected while trying to handle a prior one, and for some reason they can't be handled in turn.</td>
</tr>
<tr>
<td>9</td>
<td>Coprocessor segment overrun coprocessor_segment_overrun()</td>
<td>abort</td>
<td>SIGFPE</td>
<td>Problem with a math co-processor, like a x387 chip.</td>
</tr>
</tbody>
</table>
8.3. Interrupts

There are two kinds 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 ReQuest) line, which is connected to an APIC (Advanced Programmable Interrupt Circuit).

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    122     2     1     1     ID-APIC-edge     timer
1:     722    122     0     32     ID-APIC-edge     i8042
8:     0      0      0     0     ID-APIC-edge     rtc0
9:     9      0      0     0     ID-APIC-fastest    acpi
16:    1546   948    102    90     ID-APIC-fastest    uhub, usb0, usb1, para_marvell, nvidia
18:    16287  10     14    18141    ID-APIC-fastest    eth0, usb1, hci-bd:usb0, hci-bd:usb1, hci-bd:usb2
19:     2      1      0     0     ID-APIC-fastest    hci-bd:usb7, hci-bd:usb8
21:     0      0      0     0     ID-APIC-fastest    hci-bd:usb6
22:     534    151    2284   2284    ID-APIC-fastest    INTEL
23:     44432  38789   4444   4122    ID-APIC-fastest    hci-bd:usb5, hci-bd:usb6
28:     0      0      1     0     PCI-MSI-edge     eth1
39:    17360  2867    18160  17191    PCI-MSI-edge     hci
60:     0      0      0     0     Non-maskable interrupts
LGD:    265569 16184  147965  131046   Local timer interrupts
SPI:     0      0      0     0     Spurious interrupts
RES:    1849    1183    1520    1158    Rescheduling interrupts
CAL:     349    475    475    450     Function call interrupts
TLM:     1773    2734    1720    2689     TLM shutdowns
THM:     0      0      0     0     Thermal event interrupts
TMR:     0      0      0     0     Threshold APIC interrupts
ENR:     0      0      0     0
NIS:     0
```

Note that the numbers here are the number of times the interrupt line is 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.
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:

```c
#include <asm/irq.h>
#include <linux/interrupt.h>

void disable_irq (int irq);
void disable_irq_mmio (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_mmio()` 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:

```c
unsigned long flags;
spinlock_t my_lock;
spinlock_init (&my_lock);

spin_lock_irqsave(&my_lock,flags);
......
spin_unlock_irqrestore(&my_lock,flags);
```

You should be very careful with the use of these functions as you can paralyze the system.

---

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.
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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ_MAPPROGRESS</td>
<td>The handler for this IRQ handler is currently being executed.</td>
</tr>
<tr>
<td>IRQ_DISABLED</td>
<td>The IRQ line has been disabled.</td>
</tr>
<tr>
<td>IRQ_PENDING</td>
<td>An IRQ has occurred and been acknowledged, but not yet serviced.</td>
</tr>
<tr>
<td>IRQ_REPLAY</td>
<td>The IRQ line has been disabled but the previous occurrence on this line has not yet been acknowledged.</td>
</tr>
<tr>
<td>IRQ_AUTO_DETECT</td>
<td>The kernel is trying auto-detection on this IRQ line.</td>
</tr>
<tr>
<td>IRQ_WAITING</td>
<td>The kernel is trying auto-detection on this IRQ line and no interrupts have yet been detected.</td>
</tr>
<tr>
<td>IRQ_LEVEL</td>
<td>The IRQ line is level-triggered.</td>
</tr>
<tr>
<td>IRQ_MASKED</td>
<td>The IRQ line is masked and shouldn't be seen again.</td>
</tr>
<tr>
<td>IRQ_PER_CPU</td>
<td>The IRQ is per CPU.</td>
</tr>
</tbody>
</table>

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:

```c
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   cpu_mask *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
test {
```

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>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQF_DISABLED</td>
<td>The interrupt runs with interrupts disabled; i.e., it is a fast handler.</td>
</tr>
<tr>
<td>IRQF_SHARED</td>
<td>The IRQ may be shared with other devices, if they all mutually agree to it.</td>
</tr>
<tr>
<td>IRQF_SAMPLE_RANDOM</td>
<td>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 changed on for interrupts which arrive at predictable times.</td>
</tr>
<tr>
<td>IRQF_PROBE_SHARED</td>
<td>Set when sharing mismatches are expected to occur.</td>
</tr>
<tr>
<td>IRQF_TIMER</td>
<td>Set to indicate this is a timer interrupt handler.</td>
</tr>
<tr>
<td>IRQF_NOLACING</td>
<td>Set to exclude this interrupt from irq balancing.</td>
</tr>
<tr>
<td>IRQF_IRQMP POLL</td>
<td>Interrupt is used for polling (only the interrupt that is registered first in a shared interrupt is considered for performance reasons).</td>
</tr>
</tbody>
</table>

8.8 Installing an Interrupt Handler

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:

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

int request_irq(unsigned int irq,
                 irq_handler_t (*handler)(int irq, void *dev_id),
                 unsigned long flags,
                 const char *name,
                 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.
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.)

---

Table 8.5: IRQ handler return values

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ_NONE</td>
<td>The handler didn’t recognize the event; i.e., it was due to some other device sharing the interrupt, or it was spurious.</td>
</tr>
<tr>
<td>IRQHandled</td>
<td>The handler recognized the event and did whatever was required.</td>
</tr>
<tr>
<td>IRQ_RETVAL(x)</td>
<td>Evaluates as IRQ_HANMDLED if the argument is non-zero; IRQ_NONE otherwise.</td>
</tr>
</tbody>
</table>
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 .................................... 104
9.2 Module Licensing ...................................... 104
9.3 Automatic Loading/Unloading of Modules .......... 106
9.4 Built-in Drivers ........................................ 107
9.5 Kernel Building and Makefiles ...................... 109
9.6 Labs ................................................... 110
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 \texttt{EXPORT\_S\_YMBOL()} macro:

\begin{verbatim}
int my\_variable;
int my\_export\_fun () {
    ...
}
EXPORT\_S\_YMBOL(my\_variable);
EXPORT\_S\_YMBOL(my\_export\_fun);
\end{verbatim}

Note that the symbols will be exported even if they are declared as \texttt{static}.

It is also possible to export symbols with the macro:

\texttt{EXPORT\_S\_YMBOL\_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:

\begin{verbatim}
EXPORT\_S\_YMBOL\_M\_CPU\_S\_YMBOL();
EXPORT\_S\_YMBOL\_M\_GPU\_S\_YMBOL\_GPL();
EXPORT\_S\_YMBOL\_M\_F\_U\_T\_U\_R\_E();
EXPORT\_S\_YMBOL\_M\_S\_Y\_M\_S\_Y\_M\_GPL();
EXPORT\_S\_YMBOL\_M\_S\_Y\_M\_GPL();
\end{verbatim}

Kernels earlier than the 2.6 series exported all global symbols in a module unless they were explicitly declared as \texttt{static}; that is one reason why you see the \texttt{static} keyword so liberally used in kernel code, for the purpose of avoiding name \textit{pollution}.

Note that is still makes sense to declare symbols as \texttt{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 \texttt{static} unless there is a need to do otherwise.

9.2 Module Licensing

Modules can be licensed with the \texttt{MODULE\_LICENSE()} macro, as in:

\begin{verbatim}
MODULE\_DESCRIPTION("Does Everything");
MODULE\_AUTHOR("Vandals with Noodles");
MODULE\_LICENSE("GPL v2");
\end{verbatim}

Besides the informational content, this macro has important consequences: Any other license causes the entire kernel to be taint\textit{ed}, and warning messages appear when the module is loaded. For the most part, any system troubles, crashes etc. that arise while using a tainted kernel are most likely to be ignored by kernel developers. (The pseudo-file /\texttt{proc/sym/kernel/tainted} shows your kernel's status.)

The following licenses are understood by the kernel:

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
License & Meaning & Tainted? \\
\hline
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 text\&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 \\
\hline
\end{tabular}
\caption{Licenses}
\end{table}

You can see the licenses of loaded modules with a script like:

\begin{verbatim}
#!/bin/bash
for name in $(cat /proc/modules | awk '{print $1}');
do echo "# $name\nmodinfo $name | grep license"
done
\end{verbatim}

- A phrase sometimes heard in the Linux kernel developer community:
  \textbf{All binary modules are illegal.}
- A thorough debate on this topic was held on the kernel mailing list in late 2006. (See http://www.net/\texttt{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/mod.h>

int request_module (const 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("-m device=", 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+ nbddrv
alias char-major=254+ xycdrv

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 nbd_drv.ko and xyc drv.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] Frewing 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:

```
2.6.31: 185 #define pure_initcall(fn)     __define_initcall("",fn,0)
2.6.31: 196 #define core_initcall(fn)    __define_initcall("",fn,1)
2.6.31: 198 #define core_initcall_sync(fn) __define_initcall("",fn,ln)
2.6.31: 199 #define postcore_initcall(fn)  __define_initcall("2",fn,2)
2.6.31: 190 #define postcore_initcall_sync(fn) __define_initcall("2",fn,2n)
2.6.31: 191 #define arch_initcall(fn)      __define_initcall("3",fn,3)
2.6.31: 192 #define arch_initcall_sync(fn) __define_initcall("3",fn,3n)
2.6.31: 193 #define subsys_initcall(fn)    __define_initcall("4",fn,4)
2.6.31: 194 #define subsys_initcall_sync(fn) __define_initcall("4",fn,4n)
```

2.6.31: 196 #define fs_initcall(fn)
2.6.31: 197 #define fs_initcall_sync(fn)
2.6.31: 198 #define rootfs_initcall(fn)
2.6.31: 199 #define device_initcall_sync(fn)
2.6.31: 200 #define late_initcall(fn)
2.6.31: 201 #define late_initcall_sync(fn)

2.6.31: 202 #define _initcall(fn) device_initcall(fn)
2.6.31: 203 #define _exitcall(fn) \ 
2.6.31: 204 static exitcall_t _exitcall_##fn _exit_call - fn
2.6.31: 205 static exitcall_t _exitcall_##fn _exit_call - fn
2.6.31: 206 static exitcall_t _exitcall_##fn _exit_call - fn
2.6.31: 207 static exitcall_t _exitcall_##fn _exit_call - fn
2.6.31: 208 static exitcall_t _exitcall_##fn _exit_call - fn
2.6.31: 209 static exitcall_t _exitcall_##fn _exit_call - fn
2.6.31: 210 static exitcall_t _exitcall_##fn _exit_call - fn
2.6.31: 211 static exitcall_t _exitcall_##fn _exit_call - fn
2.6.31: 212 static exitcall_t _exitcall_##fn _exit_call - fn
2.6.31: 213 static exitcall_t _exitcall_##fn _exit_call - fn
2.6.31: 214 static exitcall_t _exitcall_##fn _exit_call - 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:

```
2.6.31: 787 static void _init do_iminitcalls(void)
2.6.31: 788 { 
2.6.31: 789     initcall_t *call;
2.6.31: 790     for (call = __early_initcall_end; call < __initcall_end; call++)
2.6.31: 791         do_one_iminitcall(call);
2.6.31: 792     for (call = __early_initcall_end; call < __initcall_end; call++)
2.6.31: 793         do_one_iminitcall(call);
2.6.31: 794     /* Make sure there is no pending stuff from the initcall sequence */
2.6.31: 795     flush_scheduled_work();
2.6.31: 796 }
```

Note that in addition to using the module_init() macro, each driver still should use the _init

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, it is quite easy. Full documentation can be found under /usr/src/linux/Documentation/Makefile.

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-c: 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:
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 lab_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/mydev
```

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 ........................................... 113
10.2 Kernel Debuggers ......................................... 116
10.3 debugfs .................................................. 118
10.4 kprobes and jprobes ..................................... 119
10.5 Labs ...................................................... 122

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).
CHAPTER 10. DEBUGGING TECHNIQUES

The information provided contains a dump of the processor registers at the time of the crash and a call trace indicating where it failed. Sometimes this is all one may need.

Getting the most 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/2008/7/7/406.

In order to cause an oops deliberately, one can do

if (diggaging_condition)
    BOR();

or

BOR_OK(diggaging_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", 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:

```c
#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);
}
```

10.1. OOPS MESSAGES

```c
module_init (my_init);
module_exit (my_exit);
```

```c
MODULE_LICENSE ("GPL v2");
```

We can disassemble the code with objdump. Doing

```bash
objdump -d crashit.ko
```

gives

```bash
crashit.ko: file format elf32-1386
```

Disassembly of section .init.text:

```bash
00000000 <init_module>:
```

```bash
0: 83 ec 00 sub 0x8, %esp
1: c7 44 24 04 00 00 00 movl 0x0, 0x4(%esp)
2: c7 44 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
1e: c7 44 24 00 00 00 00 mov 0x0, (%esp)
23: 89 44 24 04 mov 0x0, %eax
27: e8 fc ff ff ff call 28 <init_module+0x28>
2c: 31 c0 xor %eax, %eax
30: 83 c4 06 add 0x8, %esp
31: c3 ret
```

Disassembly of section .text:

```bash
00000000 <cleanup_module>:
```

```bash
0: 83 ec 08 sub 0x8, %esp
3: c7 44 24 04 00 00 00 movl 0x0, 0x4(%esp)
8: c7 44 24 00 00 00 00 movl 0x0, (%esp)
12: e8 fc ff ff ff call 13 <cleanup_module+0x13>
17: 83 c4 08 add 0x8, %esp
18: 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):

```bash
Hello: init_module loaded at address 0xf8a7000
Unable to handle kernel NULL pointer dereference at virtual address 00000000
```

```bash
Module linked in: crashit w83627hf espprom i75 i2c_sensor i2c_iis
i2c_viapro i2c_dev i2c_core smrpc binfmt_misc ucsi_boc sshd_boc
```

```bash
Printing eip: fb107f17
*pde = 00000000
Oeps: 0000 [1]
PREEMPT
```
CHAPTER 10. DEBUGGING TECHNIQUES

10.2 Kernel Debuggers

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/b-kdebug/.

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.
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 used 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:

```c
#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 points 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:

```c
ssize_t simple_read_from_buffer(void __user *to, size_t count, loff_t *ppos,
                                void __user *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:

```c
static ssize_t
sy_read(struct file *file, char *buf, size_t count, loff_t *ppos)
{
    int nbytes;
    nbytes = sprintf(kstring, "%lu", val);
    return simple_read_from_buffer(buf, count, ppos, kstring, nbytes);
}
```

Even simpler is to use the convenience functions:

```c
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);
```

There 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 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:

```c
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/334548.

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:

```c
size_t simple_read_from_buffer(void __user *to, size_t count, loff_t *ppos,
                                void __user *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:

```c
static ssize_t
sy_read(struct file *file, char *buf, size_t count, loff_t *ppos)
{
    int nbytes;
    nbytes = sprintf(kstring, "%lu", val);
    return simple_read_from_buffer(buf, count, ppos, kstring, nbytes);
}
```

Even simpler is to use the convenience functions:

```c
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);
```

There 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 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:

```c
mount -t debugfs none /sys/kernel/debug
```
10.4. KPROBES AND JPROBES

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/pt_regs.h.

The flags argument to the post handler is currently unused, and the tramp 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 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 probe 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
```

- rw------- 1 root root 0 Jun 11 08:29 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/](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.

---

10.5. Labs

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 ........................................... 125
11.2 Time Stamp Counter ............................ 127
11.3 Inserting Delays ................................. 128
11.4 What are Dynamic Timers? .................. 129
11.5 Timer Functions ............................... 129
11.6 Timer Implementation ....................... 130
11.7 High Resolution Timers ..................... 131
11.8 Using High Resolution Timers .............. 132
11.9 Labs ............................................. 135

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 introduction 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:

```c
#define time_after(a,b) \n(timeafter(a,b));
#define time_before(a,b) \n(timebefore(a,b));
#define time_eq(a,b) \n(timeeq(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

```c
#define get_jiffies_64(val)
```

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:

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

#define UNSIGNED long

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_usec (const unsigned long j);
unsigned long msecs_to_jiffies (const unsigned int n);
unsigned long usec_to_jiffies (const unsigned int n);
```

where the timespec and timeval structures should be familiar from user-space.

### 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 CPUs the clock frequency can vary continuously during operation.

The macros which obtain the value of the time stamp counter are:

```c
#include <asm/mnr.h>

rdtscl (low,high);
rdtdecl (low);
rdtsccll (val);
```

The rdtscp() macro reads the full 64-bit value of the TSC and stuffs it into two 32-bit arguments. The rdtsccll() macro gets only the lower 32 bits. This is usually sufficient (a 2 GHz system would overflow in about 2 seconds.) A third macro, rdtscclll(), 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 mnr.h header file containing the macro 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 -I1/root/2.6.24include/32) or explicitly include the macros:

```c
#define rdtscll (low,high) \n(asm volatile("rdtsc: = %0": (low), "=d": (high))
#define rdtsccl (low) \n(asm volatile("rdtsc: = %0": "=a": "=d")
#define rdtscclll (val) \n(asm volatile("rdtsc: = %0": "=a": "=d")
```

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 (High Precision Event Times) capable of very
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 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 those methods cannot go to sleep.

11.5 Timer Functions

The important data structure and functions used by kernel timers are:

```c
#include <linux/sched.h>

jiffies - jiffies + delay * Hz;
while ( time_before(jiffies, jiffies) )
{
    /* do nothing */
}
```

This is an idiotic thing to do, because jiffies is volatile, it is recored 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:

```c
#include <linux/delay.h>

void mdelay(unsigned long microseconds);
void udelay(unsigned long microseconds);
void nodelay(unsigned long milliseconds);
```

One should not expect true nanosecond resolution for mdelay(); 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:

```c
void sleep (unsigned int milliseconds);
unsigned long sleep, interruptible (unsigned int milliseconds);
```

If sleep_interruptible() returns before the sleep has finished (because of a signal) it returns the number of milliseconds left in the requested sleep period.
if the timer expires on its own.

del_timer(kwargs) 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:

```
......
in init_timer(at);
t->expires = jiffies + delay;
add_timer(&at);
```

or

```
......
mod_timer(at, 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.

11.7 HIGH RESOLUTION TIMERS

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
given time. Which list depends on the value of the expires field.

These lists are further partitioned into 5 groups:

<table>
<thead>
<tr>
<th>Group</th>
<th>Ticks</th>
<th>Time (for Hz=1000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tv1</td>
<td>&lt; 256</td>
<td>&lt; 256 nsecs</td>
</tr>
<tr>
<td>tv2</td>
<td>&lt; 2^14</td>
<td>&lt; 16.4 nsecs</td>
</tr>
<tr>
<td>tv3</td>
<td>&lt; 2^20</td>
<td>&lt; 17.5 mins</td>
</tr>
<tr>
<td>tv4</td>
<td>&lt; 2^26</td>
<td>18.6 hrs</td>
</tr>
<tr>
<td>tv5</td>
<td>&lt; &amp;infty;</td>
<td>&lt; &amp;infty;</td>
</tr>
</tbody>
</table>

The tv1 group has a vector of 256 doubly-linked lists, set up so that 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.
11.8 Using High Resolution Timers

The 

uint_t timeout_t (const long long, const unsigned long nsec); 

uint_t ktime_t ktime_add (const ktime_t k1, const ktime_t k2); 

uint_t ktime_sub (const ktime_t k1, const ktime_t k2); 

uint_t ktime_add_ns (const ktime_t k, uint64); 

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); 

struct timespec_to_ktime (const struct timespec *scale); 

struct timespec_to_ktime (const ktime_t k); 

struct timespec_to_ktime (const ktime_t k); 

w64 ktime_to_ms (const ktime_t k); 

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); 

int hrtimer_start (struct hrtimer *timer, long timeout); 

unsigned long hrtimer_forward (struct hrtimer *timer, ktime_t now, ktime_t interval); 

int hrtimer_cancel (struct hrtimer *timer); 

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 *data = container_of (my_timer, my_data, timer); 

where the first argument is a pointer to the timer structure, the second is 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 new argument should be the current time. It can be obtained with:

struct hrtimer *timer; 

ktime_t now = timer->base->get_time(); 

A timer is initialized by hrtimer_init() and is bound to the type of clock specified by which 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_NODE_ABS the argument time is absolute; if mode = HRTIMER_NODE_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:

```c
#include <linux/module.h>
#include <linux/Timer.h>
#include <linux/init.h>
#include <linux/version.h>
#include <linux/hrtimer.h>

static struct k_data
{
    struct hrtimer timer;
    htime_t period;
} *data;

static enum hrtimer_restart kfun (struct hrtimer *var)
{
    htime_t now = var->base->get_time ();
    printk(KERN_INFO "timer running at jiffies=%ld", jiffies);
    hrtimer_forward (var, now, data->period);
    return HRTIMER_NOHOLDSTART;
}

static int __init my_init (void)
{
    data = kmalloc (sizeof (*data), GFP_KERNEL);
    data->period = htime_get (1, 0); /* short period, 1 second */
    hrtimer_init (&data->timer, CLOCK_REALTIME, HRTIMER_MODE_REL);
    data->timer.function = kfun;
    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");
```

# 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 runs. (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 times 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_idhz variable) to print out the elapsed time (hopefully) more accurately.

## Lab 5: High Resolution Timers

Do the same things as in the previous exercise, creating two periodic timers, but 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.
Laboratory 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 synchronize 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 ................. 138
12.2 Atomic Operations ...................................... 139
12.3 Bit Operations ........................................... 140
12.4 Spinlocks ................................................... 141
12.5 Big Kernel Lock .......................................... 143
12.6 Mutexes ..................................................... 144
12.7 Semaphores ................................................ 145
12.8 Completion Functions .................................... 148
12.9 Reference Counts ......................................... 149
12.10 Labs ......................................................... 150
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 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.

12.2 Atomic Operations

Atomic functions (many of which are macros) are completed on a single instruction and work on a variable of type atomic_t, which is a structure defined as

```c
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:

```c
#include <asm/atomic.h>

#define ATOMIC_VAR_INIT(u) { (u) }                   
#define atomic_inc(v)  ((v)->counter)                
#define atomic_dec(v)  ((v)->counter) - 1

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_equal (atomic_t *v);  
int atomic_add_negative (int i, atomic_t *v);  
int atomic_add_return (int i, atomic_t *v);    
int atomic_add_return_equal (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_VAR_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.

Those 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:

```c
#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);
```

#### 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:

```c
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:
12.5. BIG KERNEL LOCK

On single processor systems, this is not a no-op, as the interrupt disabling and restoring still goes on, and as mentioned those 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);
...... critical code ..... rw_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 spinlocks.

There are also 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 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 in no way interacts with other code takes the BKL and suspends the other code.
Minimizing the use of the BKLI 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 BKLI 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 startup, etc.

12.6 Mutexes

A mutex (mutual exclusion 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:

```c
#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

```c
DEFVIEW_MUTEX (name);
```

at compile time or

```c
void mutex_init (struct mutex *lock);
```

at run time.

The locking primitives come in uninterruptible and interruptible forms but with only one unlocking function:

```c
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

```c
int mutex_trylock (struct mutex *lock);
```

and

```c
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. Those 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:

```c
#include <asm/semaphore.h>

void down (struct semaphore *sem);
void down_interruptible (struct semaphore *sem);
void down_write (struct rw_semaphore *sem);
void up (struct semaphore *sem);
void up_read (struct rw_semaphore *sem);
void up_write (struct rw_semaphore *sem);
```

```c
struct semaphore {
    atomic_t count;
    int sleepers;
};
```
12.7. SEMAPHORES

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

and is often initialized with the following inline convenience functions:

```c
static inline void init_MUTEX (struct semaphore *sem){
  sem_init(sem, 1);
}
static inline void init_MUTEX_LOCKED (struct semaphore *sem){
  sem_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:

```c
2.6.31.1129 SYSCALL_DEFINE2(sethostname, char __user *, name, int, len)
{
  int errno;
  char tmp[ __NEW_UTS_LEN ];
  if (tmpable(CAP_SYS_ADMIN))
    return -EPERM;
  if (len < 0 || len > __NEW_UTS_LEN)
    return -EINVAL;
  errno = -EINVAL;
  if (copy_from_user(tmp, name, len))
    struct new_utsname *u = &utsname();
  memcpy(u->nodename, tmp, len);
  memcpy(u->nodename + len, 0, sizeof(u->nodename) - len);
  errno = 0;
}
```

```c
2.6.31.1147 sys_write(Huts_sem);
```

```c
2.6.31.1149 return errno;
```

```c
2.6.31.1149 }
```
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_wait() <-> wait_for_completion()
sema_post() <-> 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 *
    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 can be identical, just give them different names.

Hint: You can use the macro STRINGIFY(BUILD_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;
made 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

ioctl

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 ioctl's? ........................................... 153
13.2 Driver Entry point for ioctl's ................................ 154
13.3 Lockless ioctl's ............................................. 155
13.4 Defining ioctl's ............................................. 156
13.5 Labs .......................................................... 158

13.1 What are ioctl's?

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() 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(), one has to first open a device using the open() system call, and then send the appropriate ioctl() command and any necessary arguments.

```c
#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>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERADD</td>
<td>Bad file descriptor.</td>
</tr>
<tr>
<td>ENOTTY</td>
<td>File descriptor not associated with a character special device, or the request does not apply to the kind of object the file descriptor refers to.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Invalid command or argp.</td>
</tr>
</tbody>
</table>

### Example:

```c
int fd = open("/dev/mydrv", O_RDWR);
if (ioctl(fd, MYDRV_SET, but) < 0)
    perror("MYDRV_SET ioctl failed");
```

#### 13.2 Driver Entry point for ioctl

The entry point for ioctl() looks like:

```c
#include <linux/ioctl.h>

static int mydrv_ioctl (struct inode *inode, struct file *file, 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:

```c
static int mydrv_ioctl (struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
{
    if (_IOC_TYPE(cmd) == MYDRV_BASE)
        return (-EINVAL);

    switch (cmd) {
    case MYDRV_RESET :
        /* do something */
        return 0;
    case MYDRV_OFFLINE :
        /* do something */
        return 0;
    case MYDRV_SELSTAY :
        if (copy_to_user((void *)&arg, &mydrv_state_struct, sizeof(mydrv_state_struct)))
            return -EFAULT;
        return 0;
    default:
        return -EINVAL;
    }
}
```

#### 13.3 Lockless ioctl

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/linear`/fs/ioctl.c.

```c
2.6.31: 37 static long vfs_ioctl(struct file *filp, unsigned int cmd, unsigned long arg) 2.6.31: 38 2.6.31: 39 { 2.6.31: 40     int error = -ENOTTY; 2.6.31: 41 2.6.31: 42     if (filp->f_op) 2.6.31: 43     goto out; 2.6.31: 44 2.6.31: 45     if (filp->f_op->unlocked_ioctl) { 2.6.31: 46         error = filp->f_op->unlocked_ioctl(filp, cmd, arg); 2.6.31: 47         if (error == -ENOIOCTLS)
```
CHAPTER 13. IOCTS

2.6.31: 48    error = -EINVAL;
2.6.31: 49    goto out;
2.6.31: 50   } else if (filp->f_op->ioctl)
2.6.31: 51       lock_kernel();
2.6.31: 52       error = filp->f_op->ioctl(filp->f_path.dentry->d_inode,
2.6.31: 53         filp, cmd, arg);
2.6.31: 54       unlock_kernel();
2.6.31: 55     }
2.6.31: 56     out:
2.6.31: 57     return error;
2.6.31: 58 }

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 waits 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 iocts

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.

13.4. DEFINING IOCTS

Two important files are /usr/src/linux/include/asyn-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>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Meaning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>IOC_TYPEBITS</td>
<td>type</td>
<td>magic number to be used throughout the driver</td>
</tr>
<tr>
<td>8</td>
<td>IOC_BRBITS</td>
<td>number</td>
<td>the sequential number</td>
</tr>
<tr>
<td>14</td>
<td>IOC_SIZEBITS</td>
<td>size</td>
<td>of the data transfer</td>
</tr>
<tr>
<td>2</td>
<td>IOC_DIRBITS</td>
<td>direction</td>
<td>of the data transfer</td>
</tr>
</tbody>
</table>

The direction can be one of the following:

- 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:

- IOC(type, number)
- IOC(type, number, size)
- IOC(type, number, size)
- IOC(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 = _IO98('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:

_IO98_DIR(cad)
_IO98_TYPE(cad)
_IO98_NR(cad)
_IO98_SIZE(cad)

Example:

#define MYDBASE 'k'
define MYDBASE_RDONLY ( MYDBASE, 1)
define MYDBASE_WRONLY ( MYDBASE, 2)
define MYDBASE_RDWR ( MYDBASE, 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 _IO98 macro directly in the user-space program. (See linux/ioct1.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 *task, int priv);

where signal is the signal to send, task 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 SIGHUP 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? ........................................ 161
14.2 Creating Entries ....................................................... 162
14.3 Reading Entries ........................................................ 163
14.4 Writing Entries .......................................................... 164
14.5 The seq_file Interface ................................................ 165
14.6 Labs ...................................................................... 167

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.
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/layer, /proc/fs,
/proc/net, /proc/stat, /proc/sock, and /proc/sock. 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 inode;
    unsigned short name_len;
    char *name;
    struct proc_dir_entry *link;
    uid_t uid;
    gid_t gid;
    unsigned long size;
    struct inode_operations *proc_ops;
    struct file_operations *proc_ops;
    struct proc_dir_entry *next, *parent, *subdir;
    void *data;
    read_proc_t *read_proc;
    }
```

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 those 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 */
    ++cct;
    return sprintf (buf, "jiffies = %d", (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.

14.5 The seq_file Interface

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
user space 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/asm/i386/vm/parsef.c:

    long simple_strtol (const char *op, char **endp, unsigned int base);
    unsigned long simple_stroull (....);
    unsigned long long simple_stroull (....);
    long long simple_strtol (....);

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_stroull ("-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/asm/proc/sequence/proc_file.h and
/usr/src/asm/proc/sequence/proc_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;
CHAPTER 14. THE PROC FILESYSTEM

```c
struct seq_operations {
    void * (start) (struct seq_file *, loff_t *) pos;
    void * (stop) (struct seq_file *, void *)
    void * (next) (struct seq_file *, void *v, loff_t *pos);
    int (*show) (struct seq_file *, 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 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

```c
static inline int seq.puts (struct seq_file *, char c);
static inline int seq.puts (struct seq_file *, 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:

```c
int seq_open (struct file *, struct seq_operations *);
size_t seq_read (struct file *, char *, size_t, loff_t *);
loff_t seq_lseek (struct file *, loff_t, int);
int seq_repos (struct inode *, struct file *);
int seq_eofpos (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:

```c
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 *, struct file *) {
    return seq_open (file, &my_seq_ops);
}
static struct file_operations my_seq_file_ops = {
    .open= my_seq_file_open,
};
```

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/pci.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:

```c
static struct seq_operations proc_bus_pci_devices_op = {
    .start= pci_seq_start,
    .next= pci_seq_next,
    .stop= pci_seq_stop,
    .show= show_device,
};
```

```c
static int proc_bus_pci_dev_open(struct inode *, struct 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,
    .lseek= seq_lseek,
    .release= seq_release,
};
```

```c
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 a permissions problem? If so try to reset them with `chmod 666 /proc/kcore` and try again.
CHAPTER 14. THE PROC FILESYSTEM

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 *task, int priv);

where signal is the signal to send, task 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, its main data structures and how they apply to real devices and examine the sysfs pseudo-filesystem.

15.1 Unified Device Model .................................. 171
15.2 Basic Structures ........................................ 172
15.3 Real Devices ............................................ 174
15.4 sysfs ..................................................... 175
15.5 Labs ..................................................... 177

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 busses.

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
15.2 Basic Structures

For every device there is a generic structure defined in /usr/src/linux/include/linux/device.h

```c
2.6.31: 367 struct device {
2.6.31: 368    struct device *parent;
2.6.31: 369
2.6.31: 370    struct device_private *p;
2.6.31: 371
2.6.31: 372    struct kobj kobj;
2.6.31: 373    const char *init_name; /* initial name of the device */
2.6.31: 374    struct device_type *type;
2.6.31: 375
2.6.31: 376    struct semaphore *sen;  /* semaphore to synchronize calls to
2.6.31: 377               * its driver.
2.6.31: 378               */
2.6.31: 379
2.6.31: 380    struct bus_type *bus;  /* type of bus device is on */
2.6.31: 381    struct device_driver *driver;  /* which driver has allocated this
2.6.31: 382         device */
2.6.31: 383    void *driver_data;  /* data private to the driver */
2.6.31: 384    void *platform_data;  /* Platform specific data, device
2.6.31: 385         core doesn’t touch it */
2.6.31: 386    struct dev_pm_info *pm;
2.6.31: 387
2.6.31: 388
2.6.31: 389 #ifdef CONFIG_KERN
2.6.31: 390 #endif
2.6.31: 391 int numa_node;  /* NUMA node this device is close to */
2.6.31: 392
2.6.31: 393 #endif
2.6.31: 394
2.6.31: 395 u64 dma_mask;  /* dmamask (if dmacheck is *)
2.6.31: 396 u64 coherent_dma_mask;  /* like dma_mask, but for
2.6.31: 397           coherent allocations as
2.6.31: 398         not all hardware supports
2.6.31: 399           64 bit addresses for consistent
2.6.31: 400         allocations each descriptors */
2.6.31: 401
2.6.31: 402 struct device_dma_parameters *dma_params;
2.6.31: 403
2.6.31: 404 struct list_head dma_pools;  /* dmamask (if dmacheck is *)
2.6.31: 405
2.6.31: 406 struct dma_coherent_pool *dma_pm;  /* internal for coherent mea
2.6.31: 407         override */
2.6.31: 408
2.6.31: 409
2.6.31: 410
2.6.31: 411
2.6.31: 412
2.6.31: 413
2.6.31: 414
2.6.31: 415
2.6.31: 416
2.6.31: 417;
```

After important fields are initialized, the device is registered with and unregistered from the system core with:

```c
2.6.31: 121 struct_device_driver {
2.6.31: 122 const char *name;
2.6.31: 123 const struct_device_type *bus;
2.6.31: 124 int (*probe)(struct_device_device *dev);  /* used for built-in modules */
2.6.31: 125 struct_module *owner;
2.6.31: 126 const char *mod_name;  /* used for built-in modules */
2.6.31: 127
2.6.31: 128
2.6.31: 129
2.6.31: 130 void (*remove)(struct_device_device *dev);
2.6.31: 131 void (*release)(struct_device_device *dev);
2.6.31: 132 int (*suspend)(struct_device_device *dev, pm_state_t state);  /* sleep */
2.6.31: 133 int (*resume)(struct_device_device *dev);
2.6.31: 134 struct_attribute_group *groups;
2.6.31: 135 void dev_pm_ops *pm;
2.6.31: 136
2.6.31: 137
2.6.31: 138;
```

Drivers are registered/unregistered with the appropriate bus with:

```c
2.6.31: 129 int driver_register (struct_device_driver *drv);
2.6.31: 130 void driver_unregister (struct_device_driver *drv);
```
and reference counts are incremented/decremented with:

```c
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:

```c
struct pci_dev {

  ...

  struct pci_driver *driver;

  ...

  struct device dev;

  ...

} ;
```

```c
struct pci_driver {

  ...

  struct device_driver driver;

  ...

} ;
```

And drivers are registered with the system with

```c
#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:

```c
struct device *dev;

...

struct pci_dev *pdev = to_pci_dev (dev);
```

which is implemented in terms of the `container_of()` macro.

```c
#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:

```c
struct device_driver *drv;

...

struct pci_driver *pdev = to_pci_dev (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 `/sys` 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/ filesystem/ 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 -1F /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 -1 /sys/class/net/eth0/
  total 0
  r--r--r-- 1 root root 4096 Jul 28 01:33 address
```
Notice that typing out the simple entries just reads out values:

```
$ cat /sys/class/net/eth0/sta
1500
```

in the way we are accustomed to getting information from the /proc filesystem. The intuition 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/devicespci0000:00
0000:00:00.0/ 0000:00:01.2/ 0000:00:1c.4/ 0000:00:1d.2/ 0000:00:1e.2/ 0000:00:1f.2/ power/
0000:00:01.0/ 0000:00:01.7/ 0000:00:1c.5/ 0000:00:1d.2/ 0000:00:1e.2/ 0000:00:1f.2/ vendor
0000:00:1a.0/ 0000:00:1b.0/ 0000:00:1c.0/ 0000:00:1d.0/ 0000:00:1e.0/ 0000:00:1f.0/ pcibus: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 use:

```
$ ls -l /sys/devicespci0000:00/0000:00:1c.0
total 0
-rw-r--r-- 1 root root 4096 Oct 15 13:27 broken_parity_status
-rw-r--r-- 1 root root 4096 Oct 15 13:27 class
-rw-r--r-- 1 root root 256 Oct 15 13:27 config
-rw-r--r-- 1 root root 4096 Oct 15 13:27 device
lrwxrwxrwx 1 root root 0 Oct 15 13:27 driver -> ../bus/pci/drivers/ekga
-rw-r--r-- 1 root root 4096 Oct 15 13:27 enable
```

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 sysfsutils package. Currently it uses libsysfs, which is being deprecating in favor of libbld.

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? ........................................... 179
16.2 Loading Firmware ........................................... 180
16.3 Labs ......................................................... 180

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 re writable 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
16.2 Loading Firmware

Loading of firmware into drivers can be done with:

```c
#include <linux/firmware.h>

struct firmware {
    size_t size;
    const char *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

```c
int dev_set_name (struct device *dev, const char *fmt, ...);
```
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 ............................................. 184
17.2 Memory Zones ............................................................. 185
17.3 Page Tables ............................................................... 186
17.4 kmalloc() ................................................................. 186
17.5 __get_free_pages() ......................................................... 188
17.6 vmalloc() ................................................................. 189
17.7 Early Allocations and bootmem() .................................... 189
17.8 Slabs and Cache Allocations ............................................ 190
17.9 Labs ................................................................. 193
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 2 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.)

![Diagram of 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 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.

Memory Zones
(32-bit x86)

<table>
<thead>
<tr>
<th>Zone</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>HIGH</td>
<td>896 MB</td>
</tr>
<tr>
<td>NORMAL</td>
<td>16 MB</td>
</tr>
<tr>
<td>DMA</td>
<td>0 MB</td>
</tr>
</tbody>
</table>

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
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 PPN into a physical one, using page tables. Each entry in the page table contains a valid flag, the PPN, 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:

```c
#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>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFP_KERNEL</td>
<td>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().</td>
</tr>
<tr>
<td>GFP_ATOMIC</td>
<td>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.</td>
</tr>
<tr>
<td>GFP_DMA</td>
<td>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 MALLOCABLE_ADDRESS=16 MB on x86 for ISA devices; for PCI this is unnecessary. The exact meaning of this flag is platform dependent.</td>
</tr>
<tr>
<td>GFP_USER</td>
<td>Used to allocate memory for a user. May sleep, and is a low priority request.</td>
</tr>
<tr>
<td>GFP_HUGGER</td>
<td>Like GFP_USER, but allocates from high memory</td>
</tr>
<tr>
<td>GFP_NOIO</td>
<td>Not to be used for filesystem calls, disallows I/O initiation.</td>
</tr>
<tr>
<td>GFP_HFS</td>
<td>For internal use.</td>
</tr>
</tbody>
</table>

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:

```c
char *buffer = kmalloc (bytes, in_interrupt() ? GFP_ATOMIC : GFP_KERNEL);
```

A similar macro, `in_atomic()`, also checks to see if you are in a preemtable 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:

```c
void *kmalloc (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:

```c
void *kresalloc (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 KiB) it is better to use the __get_free_pages() functions.

### 17.5 __get_free_pages()

To allocate (and free) entire pages (or multiple pages) of memory in one fell swoop one can use:

```c
#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:

```c
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:

```c
#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 contiguous 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:

```c
in_buf[dev]= (struct mbuf *)vmalloc(siswof(struct mbuf));
if (in_buf[dev] == NULL)
    {
        printk(KERN_WARNING "Can’t allocate buffer in_buf\n");
        my_dev[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:

```c
#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.
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, for a task, and add and remove objects from it (all of the same size) as needed.

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 the task runs 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:

```c
#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 (1 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>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLAB_MAIN</td>
<td>Force alignment of data objects on cache lines. This improves performance but may waste memory. Should be set for critical performance code.</td>
</tr>
<tr>
<td>SLAB_AUTO</td>
<td>Fill the slab layer with the known value, aSaSa5a5. Good for caching access to uninitialized memory.</td>
</tr>
<tr>
<td>SLAB_NOحال</td>
<td>Surround allocated memory with hal zones that scream when touched, to detect buffer overruns.</td>
</tr>
<tr>
<td>SLAB_PANIC</td>
<td>Cause system panic upon allocation failure.</td>
</tr>
<tr>
<td>SLAB_DEBDU_FREE</td>
<td>Perform expensive checks on freeing objects.</td>
</tr>
<tr>
<td>SLAB_CACHE_DMA</td>
<td>Make sure the allocation is in the DMA zone.</td>
</tr>
</tbody>
</table>

When your cache has been deployed, name will show up under /proc/slabinfo, and will show something like:

```
slabinfo - version: 1.1
ksme_cache   59 78 100  2  2  1
miscache     0  1 4096  0  1  1
ip_comtrack  0 11 382  0  1  1
tcp_v3_bucket  0  0  96  0  0  1
tcp_v4_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
CHAPTER 17. MEMORY MANAGEMENT AND ALLOCATION

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 `vmscan -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:

```c
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 `kfree_pages()`. (If the memory doesn’t already exist in the cache, it will be created using those 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.

---

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 `malloc()` and `get_free_pages()`.

Start with requesting 1 page of memory, and then keep doubling until your request fails for each type of allocation.

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, try 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 `malloc()`. 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 ........................................ 196
18.2 put(get)_user() and copy_to/from)_user() ..................... 196
18.3 Direct transfer - Kernel I/O and Memory Mapping .......... 198
18.4 Kernel I/O .......................................................... 199
18.5 Mapping User Pages .................................................. 200
18.6 Memory Mapping ....................................................... 201
18.7 User-Space Functions for mmap() ................................. 202
18.8 Driver Entry Point for mmap() ..................................... 204
18.9 Relay Channels ....................................................... 207
18.10 Relay API ............................................................. 208
18.11 Accessing Files from the Kernel .................................. 209
18.12 Labs ................................................................. 212
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.

```c
#include <linux/access.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 __strlen_from_user (char *src, const char *src, long count);
long strlen_user (const char *str);
long strlen_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.
CHAPTER 18. TRANSFERRING BETWEEN USER AND KERNEL SPACE

18.4. KERNEL I/O

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 userspace 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.)

```
/*
 * arg 1 = input file, 2 = output file, [3 = chunk size]
 * usage: x copy_ofile
 *        %x copy_ofile 64K
 */

#define _GNU_SOURCE
#define SECTOR_SIZE 512

#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#define __alloc__
#define __string
int main (int argc, char *argv[])
{
```
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:

```c
#include <linux/mm.h>

int get_user_pages(struct task_struct *task,  
                   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.

18.6 Memory Mapping

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:

```c
down_read(current->mm->map_sem);
rc = get_user_pages(current, current->mm, (unsigned long) buf, npages, 0, 0, pages, NULL);
up_read(current->mm->map_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:

```c
#include <linux/page.h>
```

```c
... char *kbuf = page_address(pages[i]);
```

or do

```c
char *kbuf = kmap(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 page; 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:

```c
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.
18.7 User-Space Functions for mmap()

From the user side, memory mapping is done with:

```c
#include <unistd.h>
#include <sys/mman.h>

void *mmap (void *start, size_t length, int prot, int flags, int fd, off_t offset);
int mmunmap (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>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROT_NONE</td>
<td>Pages may not be accessed.</td>
</tr>
<tr>
<td>PROT_WRITE</td>
<td>Pages may be written.</td>
</tr>
<tr>
<td>PROT_READ</td>
<td>Pages may be read.</td>
</tr>
<tr>
<td>PROT_EXEC</td>
<td>Pages may be executed.</td>
</tr>
</tbody>
</table>

Table 18.2: mmap() memory protection bits

flags specifies the type of mapped object. It has bits:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAP_FIXED</td>
<td>If start can’t be used, fail.</td>
</tr>
<tr>
<td>MAP_SHARED</td>
<td>Share the mapping with all other processes.</td>
</tr>
<tr>
<td>MAP_PRIVATE</td>
<td>Create a private copy-on-write mapping.</td>
</tr>
</tbody>
</table>

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:

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.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);
    }
}```
18.8. Driver Entry Point for `mmap()' 

From the kernel side, the driver entry point looks like:

```c
#include <linux/mm.h>

int (*remap)(struct file *filp, struct vm_area_struct *vma);
```

The `vm_area_struct` data structure is defined in `/usr/src/linux/include/linux/mm.h` and contains the important information. The basic elements are:

```c
struct vm_area_struct {
...
    __u64 __vm_start;  /* The start address within vm_mm. */
    __u64 __vm_end;    /* The first byte after our end address within vm_mm. */
    __u64 __pgprot;    /* Access permissions of this VMA. */
    __u64 __vm_flags;  /* Flags, listed below. */
    ...  
    __u64 __vm_ops;    /* Function pointers to deal with this struct. */
    struct vm_operations_struct * __vm_ops;
    ...  
    __u64 __vm_pageoff;  /* 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()`, `advise()`, `swapout()`, `swaupin()`, ...

A simple example serves to show how the fields are used:

```c
#include <linux/mm.h>

int my_remap (struct file *file, struct vm_area_struct *vma)
{
    if (remap_pfn_range (vma, vma->vm_start, vma->vm_end))
        return -ENOSPC;
    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:

```c
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <string.
#include <errno.h>
#include <sys/mman.

#define MMAP(msg) (perror(msg); exit(errno));
```

```c
int main (int argc, char **argv)
{
    int fd, size, rc, j;
    char *argv, *cmd, *nodename = "raw/mymap";
    char c[2] = "CI"
    if (argc > 1)
    {
        nodename = argv[1];
    }
    if (argc > 2)
    {
        size = atoi(argv[2]);
    }
    if ((fd = open (nodename, O_RDWR)) < 0)
    {
        perror ("Error creating the node ");
    }
    mmap (NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (area == MAP_FAILED)
    {
        perror ("Error mapping ");
    }
    close (fd);
    if (write (STDOUT_FILENO, area, size))
    {
        put ("The thing is not working ");
    }
    return 0;
}
```
Here is a simple driver with a `map()` entry point:

```c
#include <linux/module.h>  // for modules *
#include <linux/fs.h>     // file_operations */
#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 "mydev"

static dev_t first;  // static dev_maj my_major = 700; my_minor = 0;
static struct cdev my_cdev;
static struct file_operations mydev_fops = {
    owner = THIS_MODULE,
    map = mydev_map,
};

static int _init my_init (void) {
    first = DEVIDE(my_major, my_minor);
    register_chrdev_region(first, count, MYDEV_NAME);
    my_cdev = cdev_alloc();
    cdev_init (my_cdev, MYDEV_NAME);
    cdev_add (my_cdev, first, count);
    printk(KERN_INFO "unsuccessed in registering character device \n", MYDEV_NAME);
    return 0;
}

module_init (my_init);
module_exit (my_exit);
```

Module Автор: "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. 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 `mmmap()`, 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 `mmmap()` 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()` Consumes 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.

- `mmmap()`, `unmap()` 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, POLLOUT, POLLRDNORM, POLLRDokane 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

```c
#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 *rchan);
```

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 file 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:

```c
void relay_write(struct rchan *rchan, 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:

```c
struct rchan_callbacks {
  int (*subbuf_start)(struct rchan_buf *buf,
                      void *subbuf,
                      size_t prev_subbuf,
                      size_t prev_padding);
  void (*buf_mapped)(struct rchan_buf *buf,
                     struct file *file);
  void (*buf_unmapped)(struct rchan_buf *buf,
                        struct file *file);
  struct dentry *(*create_buf_file)(const char *filename,
                                     struct dentry *parent,
                                     int mode);
  struct rchan_buf *(*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 in_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:

```c
static struct dentry
*create_buf_file_handler(const char *filename,
                         struct dentry *parent,
                         int mode,
                         struct rchan_buf *buf,
                         int in_global)
{
    return debugfs.create_file(filename, mode, parent, buf,
                      &relay_file_operations);
}
```

or:

```c
static int remove_buf_file_handler(struct dentry *dentry)
{
    debugfs.remove(dentry);
    return 0;
}
```

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/2005-23/1447.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 owner; 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 macro for handling this are:

```c
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/user.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:

```c
#include <linux/module.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/fs.h>

static char *filename = "/tmp/tmpfile";
static struct file *file;
static long offset = 0;

int kernel_read(struct file *file, char *buffer, long count)
{
    struct vfs_file *vfs_file = file;
    struct file *file;
    struct page *page;

    return vfs_read(vfs_file, buffer, count, offset);
}

int kernel_write(struct file *file, char *buffer, long count)
{
    struct vfs_file *vfs_file = file;
    struct file *file;
    struct page *page;

    return vfs_write(vfs_file, buffer, count, offset);
}
```

```c
#define NYBYES_TO_READ 20

static int __init my_init(void)
{
    struct file *file;
    int bytes,
    char *buffer;

    buffer = kmalloc(PAGE_SIZE, GFP_KERNEL);
    printk(KERN_INFO "Opening file %s", filename);
    if (filp_open(filename, O_RDWR, S_IRUSR | S_IWUSR))
        return -EIO;

    if (IS_ERR(buffer))
        printk(KERN_INFO "Error opening %s", filename);

    memset(buffer, 0, bytes);
    f->f_pos += bytes;
    f->f_pos &= ~7;
    f->f_op->write(f, buffer, bytes);

    return 0;
}

static void __exit my_exit(void)
{
    printk(KERN_INFO "Exiting %s", filename);
}
```

Such a method should never be used in code that is submitted to the kernel.
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 map() 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? ......................... 213
19.2 Going to Sleep and Waking Up ................... 214
19.3 Going to Sleep Details ......................... 216
19.4 Exclusive Sleeping ............................ 218
19.5 Waking Up Details .................. ............ 218
19.6 Polling ........................................ 220
19.7 Interrupt Handling in User-Space .............. 221
19.8 Labs ..................................... 222

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 met. 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 sleep and waking up functions come in two forms, interruptible and non-interruptible. Non-interruptible 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 if the 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 resumed (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 the statements

```c
#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

```c
DECLARE_WAITQUEUE(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

```c
#include <linux/wait.h>
```

```c
void wake_up()

void wake_up_interruptible()
```

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 `ESTAB` 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, as such 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 were woken up because a signal arrived, or there was an explicit wake up call. The `signal_pend` macro can be used for this purpose.

The condition test has two important purposes:

- It checks if the condition is true and the task is woken up as a result.
- It checks if the condition is true and the task is woken up as a result.

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 to true (non-zero).

Sometimes you want to ensure you don't sleep too long. For this purpose one can use:

```c
void wake_up_interruptible_timeout()

void wake_up_interruptible_timeout()
```

where the timeout is specified in jiffies. If the task returns before 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 `ESTAB`.

The waking functions will remove 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:

```c
void wake_up_interruptible_timeout()
```

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 fn1 (...) {
...
    printk(KERN_INFO "task %u (%s) going to sleep\n", current->pid, current->comm);
    wait_event_interruptible(wq, datarady);
    printk(KERN_INFO "awoken %u (%s)\n", current->pid, current->comm);
    if (signal_pending(current))
        return -ESTARTSYS;
    ...
    datarady = 0;
}
static int fn2 (...) {
...
    printk(KERN_INFO "task %u (%s) awakening sleepers...\n", current->pid, current->comm);
    datarady = 1;
    wake_up_interruptible(&wq);
    ...
}

(Note the variable datarady 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 how a wait, or going to sleep. The macro `wait_event()` is defined in `/usr/src/linux/include/linux/wait.h`.

2.6.31: 198 #define wait_event(wq, condition)
2.6.31: 199 do {
2.6.31: 199   if (condition)
2.6.31: 200     break;
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   for (;;) {
2.6.31: 176     prepare_to_wait(&wq, __wait, TASK_INTERRUPTIBLE);
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)

19.3. Going to Sleep Details

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()` or `wait_event()`, 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_entry_t *wq, 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_INTERRUPTIBLE`.

The next line 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_entry_t *wq, 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: 256    for (;;) {
2.6.31: 257      prepare_to_wait(&wq, __wait, TASK_INTERRUPTIBLE);
2.6.31: 258      if (condition)
2.6.31: 259      break;
2.6.31: 261      if (!signal_pending(current)) {
2.6.31: 262      schedule();
2.6.31: 264      continue;
2.6.31: 266      re->ESTARTSYS;
2.6.31: 268      break;
2.6.31: 270    }
2.6.31: 272    finish_wait(&wq, __wait);
2.6.31: 274 } while (0)

which checks to see if the sleep ended because of an incoming signal, and if so returns the value `ESTARTSYS`.

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:

\[\text{wait_event_interruptible_exclusive(wait_queue_head_t wq, int condition);}\]

(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:

\[\text{void wake_up_all(wait_queue_head_t *wq);}\]
\[\text{void wake_up_interruptible_all(wait_queue_head_t *wq);}\]
\[\text{void wake_up_nr(wait_queue_head_t *wq, int nr);}\]
\[\text{void wake_up_sync_nr(wait_queue_head_t *wq, int nr);}\]
\[\text{void wake_up_interruptible_sync_nr(wait_queue_head_t *wq, int nr);}\]

The ones with all in the name wake up all tasks in the queue, just as in the non-exclusive case, but those with \textit{nr} awaken only \textit{nr} tasks (typically \textit{nr}=1.)

19.5 Waking Up Details

All the wake up calls are macros that invoke the basic \texttt{__wake_up()} call and are defined in \texttt{/usr/src/linux/include/linux/wait.h}:

2.6.31: 149 \#define wake_up(x)
2.6.31: 150 \#define wake_up_nr(x, nr)

\[\text{__wake_up(x, TASK_NORMAL, 1, NULL);}\]
\[\text{__wake_up(x, TASK_NORMAL, nr, NULL);}\]

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 (\texttt{curr->func()}) which by default is set to \texttt{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 \texttt{try_to_wake_up()}:
CHAPTER 19. SLEEPING AND WAIT QUEUES

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 mydrv_fops = {
    .owner = THIS_MODULE,
    ...
    .poll = mydrv_poll,
};

static unsigned int mydrv_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_hand_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>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>POLLIN</td>
<td>Normal or priority bandwidth data can be read without blocking.</td>
</tr>
<tr>
<td>POLLIN</td>
<td>Normal data can be read. Usually a readable device returns POLLIN</td>
</tr>
<tr>
<td>POLLIN</td>
<td>Priority bandwidth data can be read. (This flag is unused.)</td>
</tr>
<tr>
<td>POLLIN</td>
<td>High priority out of band data can be read, causing select() to report</td>
</tr>
<tr>
<td>POLLIN</td>
<td>an exception.</td>
</tr>
<tr>
<td>POLLIN</td>
<td>Reaching end of file on device.</td>
</tr>
<tr>
<td>POLLIN</td>
<td>An error has occurred.</td>
</tr>
<tr>
<td>POLLIN</td>
<td>The device can be written without blocking.</td>
</tr>
<tr>
<td>POLLIN</td>
<td>Normal data can be written. Usually a writable device returns POLLOUT</td>
</tr>
<tr>
<td>POLLIN</td>
<td>Priority bandwidth data can be written.</td>
</tr>
</tbody>
</table>

An example of an entry point might look like:

static unsigned int mydrv_poll (struct file *filp, poll_table *wait)
{
    unsigned int revents = 0;
    poll_wait (filp, &wq_read, wait);
    poll_wait (filp, &wq_write, wait);

    if (atomic_read (&data_ready_to_read))
        revents |= POLLIN | POLLIN;
    if (atomic_read (&data_ready_to_write))
        revents |= POLLOUT | POLLOUT;
    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 isopl() and isosern() commands to get application access to I/O ports, or are layered on top of 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 device 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/127898/ 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 asynchronosness 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.

19.8 Labs

Lab 1: Using Wait Queues
Generalize the previous character driver to use wait queues,
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/ttydev 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, focusing 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 ........................................... 225
20.2 Deferrable Functions and softirqs ......................... 227
20.3 Tasklets .......................................................... 228
20.4 Work Queues ..................................................... 231
20.5 Creating Kernel Threads ......................................... 234
20.6 Threaded Interrupt Handlers ................................... 235
20.7 Labs ............................................................... 235

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):**

```c
static struct my_dat { .... } my_fem_data;
static void t_fem (unsigned long t_arg){ .... }
DECLARE_TASKLET (t_name, t_fem, (unsigned long) &my_data);
static void my_interrupt (int irq, void *dev_id)
{  
    top_half_fem ();  
    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.

**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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Priority</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>HI_SOFTIRQ</td>
<td>0</td>
<td>High-priority tasklets.</td>
</tr>
<tr>
<td>TIMER_SOFTIRQ</td>
<td>1</td>
<td>Scheduled timers.</td>
</tr>
<tr>
<td>NET_TX_SOFTIRQ</td>
<td>2</td>
<td>Network packet transmission.</td>
</tr>
<tr>
<td>NET_RX_SOFTIRQ</td>
<td>3</td>
<td>Network packet reception.</td>
</tr>
<tr>
<td>BLOCK_SOFTIRQ</td>
<td>4</td>
<td>Block device related work.</td>
</tr>
</tbody>
</table>
### Tasklets

Tasklets may be run in parallel on multiple CPU systems. However, the same tasklet cannot 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:

```c
struct tasklet_struct
{
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    void (*func)(unsigned long);
}
```

The `func` entry is a pointer to the function that will be run, which can have data passed to it through a data. The `state` entry is used to determine whether or not the tasklet has already been scheduled; if so it cannot be done so a second time.

The main macros and functions involving tasklets are:

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

```c
void tasklet_init (struct tasklet_struct *,
    void (*func)(unsigned long), unsigned long data);
```

```c
void tasklet_schedule (struct tasklet_struct *);
```

```c
void tasklet_enable (struct tasklet_struct *);
```

```c
void tasklet_disable (struct tasklet_struct *);
```

```c
void tasklet_kill (struct tasklet_struct *);
```

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 reach themselves.

When a tasklet is scheduled, the `inline` function `tasklet_schedule()` is called as defined in `/usr/src/linux/include/linux/interrupt.h`:

```c
2.6.31: 404 static inline void tasklet_schedule(struct tasklet_struct *t)
2.6.31: 405 {
    2.6.31: 406 if (!(test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
    2.6.31: 407    tasklet_schedule(t);
    2.6.31: 408 }
```
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:

```c
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
#include <linux/slab.h>
#include <linux/init.h>

static void t_fum (unsigned long t_arg);
static struct simp {
  int i;
  int j;
} t_data;

static DECLARE_TASKLET (t_name, t_fum, (unsigned long)t_data);

static int __init my_init (void) {
  printk(KERN_INFO "\Hello, 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= Xld \n", jiffies);
  tasklet_schedule (&t_name);
  return 0;
}

static void __exit my_exit (void) {
  printk(KERN_INFO "\Hello, cleanup_module loaded at address 0x%p\n",
          cleanup_module);
}

static void t_fum (unsigned long t_arg) {
  struct simp *tdata = (struct simp *)t_arg;
  printk(KERN_INFO "Entering t_fum, datum->i = \x5, jiffies = Xld\n",
          datum->i, jiffies);
  printk(KERN_INFO "Entering t_fum, datum->j = \x5, jiffies = Xld\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:
CHAPTER 20. INTERRUPT HANDLING AND DEFFERABLE FUNCTIONS

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 attach entry for tasks; 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 work queues 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 my_func (struct work_struct *w_arg)
{
    struct my_dat *data = container_of (w_arg, struct my_dat, work);
    atomic_inc (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:

20.4. WORK QUEUES

int schedule_work (struct work_struct *work);
void flush_work (void);

flush_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 *w);

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_context (struct work_struct *work);

int execute_in_process_context (work_func_t fn, struct execute_context *w);

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(sav_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_init (struct task_struct *, 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:

```
dc { .... } 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 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_NOREQ; 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 areas. 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" by 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 busses 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 .............................................. 240
21.2 Memory Barriers ........................................... 240
21.3 Registering I/O Ports ..................................... 241
21.4 Resource Management .................................... 242
21.5 Reading and Writing Data from I/O Registers ...... 244
21.6 Slowing I/O Calls to the Hardware .................... 245
21.7 Allocating and Mapping I/O Memory ................. 246
21.8 Accessing I/O Memory .................................... 247
21.9 Access by User - ioperm(), iopl(), /dev/port ...... 249
21.10 Labs ....................................................... 249
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 addressed 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 22-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.

21.3 Registering I/O Ports

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:

```c
void barrier (void)
void rmb (void)
void wb (void)
void nb (void)
void smp_rmb (void)
void smp_wb (void)
void smp_nb (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; wb() does the same thing for writes, while nb() 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:

```c
io32write (direction, dev->base + OFF_EIR);
io32write (size, dev->base + OFF_SIZE);
wb();
io32write (value, dev->base + OFF_SD);
```

Most architectures define convenience macros, which combine setting a value with invoking a memory barrier. In the simplest form they look like:

```c
#define set_m(val) (val, value; nb(); )
#define set_wb(val) (val, value; wb(); )
#define set_rmb(val) (val, value; rmb(); )
```

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 sb() if wb() 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:
21.4 Resource Management

You may have noticed that the request_region() function returns a pointer to a structure of type:

```c
struct resource {
    const char *name;
    unsigned long start, end;
    unsigned long flags;
    struct resource *parent, *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 iport_resource head node. Management of the I/O ports can be done through the functions:

```c
#include <linux/iport.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:

```c
#include <linux/iport.h>

static struct resource my_resource = { "my_dev", 0);

static int my_dev_detect( unsigned long port_addr,
                          unsigned long extent )
{
    if ( request_region( port_addr, extent, "my_dev")
        return -BUSY ; /* the port is busy */
    if ( mydrv_probe(port_addr,extent) != 0 )
        return -ENODEV /* can't find the device */
    return 0 ;
}
```

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:**

```c
unsigned char inb (unsigned long port_address);
unsigned short inw (unsigned long port_address);
unsigned long inl (unsigned long port_address);
void inb (unsigned long port_address, void *addr, unsigned long count);
void inw (unsigned long port_address, void *addr, unsigned long count);
void inl (unsigned long port_address, void *addr, unsigned long count);
```

**Writing:**

```c
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 outb (unsigned port_address, void *addr, unsigned long count);
void outw (unsigned port_address, void *addr, unsigned long count);
void outl (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.

21.6 Slowing I/O Calls to the Hardware

Reading and writing I/O ports may require the use of memory barriers, which we previously discussed.

**Example:**

```c
outb(MSR_READ_XLOW, MSR_CONTROL_PORT);

dx = (inb(MSR_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(), outb_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:

```c
/* IPORT_FROM 0x200 to 0x240 is free on my system (64 bytes) */
#define INPORT 0x200
#define IOEVENT 0x240

#include <linux/module.h>
#include <linux/ioport.h>
#include <linux/jiffies.h>
#include <linux/io.h>
#include <linux/init.h>

#define KLOOP 1000000000
#define BILL 1000000000000

/* should be a multiple of millions */

static int __init my_init (void)
{
    int jiff;
    unsigned long ulsstart = (unsigned long)1000;
    unsigned long jiff, jiffc, jifff;
    if ((request_region (IOEVENT, IOEVENT, "my_ioport")) {  
        printk(KERN_INFO "the IO region is busy, quitting\n");
        return -ENOY;
    }
    printk(KERN_INFO " requesting the IO region from 0x%lx to 0x%lx",  
           IOEVENT, IOEVENT + IOEVENT);

    /* get output delays */
    jiff = jiffc;
```
CHAPTER 21. HARDWARE I/O

21.7 Allocating and Mapping I/O Memory

Non-trivial peripheral devices are almost always accessed through 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 dependency by using a universal interface. While some architectures permit direct dereferencing of pointers for these regions, one should never attempt this.

21.8 Accessing I/O Memory

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:

```c
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/ioports. 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 unmmap it with:

```c
#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:

```c
#include <asm-generic/iomap.h>
void iomap (unsigned long port, unsigned int count);
void iomap (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

```c
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/ibb/ioapic.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:

```c
#include <linux/io.h>
```
CHAPTER 21. HARDWARE I/O

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()
funtions 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 memmove.io (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 non... 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 readdl (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 iop10(), 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(), inout() 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>
<thead>
<tr>
<th>Port</th>
<th>Node</th>
<th>IRQ</th>
<th>IOREG</th>
</tr>
</thead>
<tbody>
<tr>
<td>com1</td>
<td>/dev/ttyS0</td>
<td>4</td>
<td>0x03f8-0x03ff</td>
</tr>
<tr>
<td>com2</td>
<td>/dev/ttyS1</td>
<td>3</td>
<td>0x02f8-0x02ff</td>
</tr>
<tr>
<td>com3</td>
<td>/dev/ttyS2</td>
<td>4</td>
<td>0x03f8-0x03ff</td>
</tr>
<tr>
<td>com4</td>
<td>/dev/ttyS3</td>
<td>3</td>
<td>0x02f8-0x02ff</td>
</tr>
</tbody>
</table>

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 the 8-bit device as 3-byte packets. The dx and dy movements are sent as two's-complement, lb (rh) are set when the left (right) button is pressed:

```
byte d8 d5 d4 d3 d2 d1 d0
1 1 lb rb dy7 dy6 dy5 dy4 dy3 dy2 dx2 dx1 dx0
2 0 dx6 dx5 dx4 dx3 dx2 dx1 dx0
3 0 dy5 dy4 dy3 dy2 dy1 dy0
```

You will also have to take a 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 -D -t xs = /dev/ttyS0
```

and then kill it, which should set things up ok. (On some PCs 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 setupserial or, depending on how you handle the next step, merely opening /dev/ttyS0 from a user-space application. You can also try

```
stty -F /dev/ttyS0 speed 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 zeroes 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? ................................................. 253
22.2 PCI Device Drivers ........................................ 256
22.3 PCI Structures and Functions ......................... 258
22.4 Accessing Configuration Space .................... 259
22.5 Accessing I/O and Memory Spaces ............... 260
22.6 PCI Express ................................................. 261
22.7 Labs ..................................................... 261

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.
Information on the PCI devices currently installed on the system can be obtained with the command:

```
$ lpci -v
```

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>
<thead>
<tr>
<th>Table 22.1: PCI features</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bus number:</strong></td>
</tr>
<tr>
<td><strong>device number:</strong></td>
</tr>
<tr>
<td><strong>function number:</strong></td>
</tr>
</tbody>
</table>

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:
22.2 PCI Device Drivers

One registers and unregisters a PCI device driver with:

```c
#include <linux/pci.h>

int pcc_register_driver (struct pcc_driver *);
void pcc_unregister_driver (struct pcc_driver *);
```

The registration/de-registration functions are normally called in your initialization and cleanup functions.

The function pcc_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 pcc_driver:

```c
2.6.31: 473 struct pcc_driver {
2.6.31: 474 struct list_head node;
2.6.31: 475 char *name;
2.6.31: 476 const struct pcc_device_id *id_table; /* must be non-NULL for probe to be called */
2.6.31: 477 int (*probe) (struct pcc_dev *dev, const struct pcc_device_id *id); /* New device inserted */
2.6.31: 478 void (*remove) (struct pcc_dev *dev); /* Device removed (NULL if not a hotplug capable driver) */
2.6.31: 479 int (*suspend) (struct pcc_dev *dev, pm_message_t state); /* Device suspended */
2.6.31: 480 int (*suspend_end) (struct pcc_dev *dev, pm_message_t state);
2.6.31: 481 int (*resume) (struct pcc_dev *dev); /* Device woken up */
2.6.31: 482 void (*shutdown) (struct pcc_dev *dev);
2.6.31: 483 struct pcc_error_handlers *err_handler;
2.6.31: 484 struct pcc_driver *driver;
2.6.31: 485 struct pcc_data data;
2.6.31: 487);
```

Important elements of the data structure are:

- `id_table`: Points to a table of device IDs of interest to the driver. Usually this will be exported with the macro MDL_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 pcc_device_id which must end with a NULL entry:

```c
2.6.31: 17 struct pcc_device_id {
2.6.31: 18  __u32 vendor, device;
2.6.31: 19  __u32 subvendor, subdevice;
2.6.31: 20  __u32 class, class_mask;
2.6.31: 21  u64 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:

```c
static struct pcc_device_id sgl_pci_tbl[] = {
  PCI_DEVICE(PCI_VEND_ID_BROADCOM, PCI_DEVICE_ID_BROADCOM_5700),
  PCI_DEVICE(PCI_VEND_ID_BROADCOM, PCI_DEVICE_ID_BROADCOM_5701),
  .....
  PCI_DEVICE(PCI_VEND_ID_ALTIMA, PCI_DEVICE_ID_ALTIMA_A20100),
  PCI_DEVICE(PCI_VEND_ID_APPLE, PCI_DEVICE_ID_APPLE_T1380),
};
```

This 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 it was in suspended state. Usually this function would be called from the probe callback function.
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:

```c
#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:

```c
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 hot-plug-able.

22.4 Accessing Configuration Space

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:

```c
#include <linux/pci.h>
struct pci_dev *pdev = NULL;
if (!pdev = pci_get_device(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_8132, pdev))
    return -ENOENT;
pci_enable_device(pdev);
```

If desired, the function

```c
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:

```c
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.

```c
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.,

```
pcl_read_config_byte(pdev, 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 *pdev, int bar);`
- `unsigned long pci_resource_end (struct pci_dev *pdev, int bar);`
- `unsigned long pci_resource_len (struct pci_dev *pdev, int bar);`
- `unsigned long pci_resource_flags (struct pci_dev *pdev, int bar);`

in which `bar` stands for Bus 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/ioprobe.h.
```

Here’s an example of usage from /usr/src/linux/drivers/net/8139too.c:

```c
2.6.31: 728 static __devinit struct net_device *x8139too_init_board (struct pci_dev *pdev)
2.6.31: 729 {   ...
2.6.31: 760 rc = pci_enable_device(pdev);
2.6.31: 761 if (!rc)
2.6.31: 762 goto err_out;
2.6.31: 763 ...
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_base = pci_resource_base(pdev, 0);
2.6.31: 771 pio_mem = pci_resource_len (pdev, 0);
```

## 22.6 PCI Express

PCI was introduced in 2002 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 (Accelerated Graphics Port) 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 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 at 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 `pciuutils` 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 -hlp.
For example, to get very verbose information about all Intel devices on your system you could
type lspci -vvv -d 0x8086:*; or for AMD devices, lspci -vvv -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 -v -d 0x1022:*;
DEVICE_ID, where the -B 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/pcc_regs.h and /usr/src/linux/include/linux/pcc_ids.h
to get symbolic names.) Fields you may wish to obtain could include: PCC_VENDOR_ID, PCC_DEVICE_ID,
PCC_REVISION_ID, PCC_INTERRUPT_LINE, PCC_LATENCY_TIMER, PCC_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? ........................................ 264
23.2 DMA and Interrupts .................................. 264
23.3 DMA Memory Constraints ............................ 265
23.4 DMA Directly to User ................................ 266
23.5 DMA under PCI ...................................... 266
23.6 DMA Pools .......................................... 269
23.7 Scatter/Gather Mappings ............................. 269
23.8 DMA under ISA ..................................... 271
23.9 Labs ................................................. 272
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.

23.3 DMA Memory Constraints

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 malloc(), only malloc() 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:

```c
#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 in Linux 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:

```c
#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:

```c
struct pci_dev *pdev;
if ( !dma_supported(pdev->dev, 0xffffffff) ) {
    printk(KERN_WARNING "DMA not supported for the device\n");
    goto device_unsupported;
} else {
    /* DMA is supported. */
}
```

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:

```c
void dma_alloc_coherent (struct device *dev, size_t size, dma_addr_t *dev_addr, gfp_t flag);
void dma_free_coherent (struct device *dev, size_t size, void *vaddr, dma_addr_t dma_addr);
```

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 asynchronous 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:

```c
void 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-compatible fashion; i.e., contiguous and in the right address range. The direction argument can have the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCI_DMA_TODEVICE</td>
<td>Data going to the device, e.g., a write.</td>
</tr>
<tr>
<td>PCI_DMA_FROMDEVICE</td>
<td>Data coming from the device, e.g., a read.</td>
</tr>
<tr>
<td>PCI_DMA_BIDIRECTIONAL</td>
<td>Data going either way.</td>
</tr>
<tr>
<td>PCI_DMA_NONE</td>
<td>Used for debugging; any attempt to use the memory causes a crash.</td>
</tr>
</tbody>
</table>
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 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:

```c
#include <linux/pci.h>
int pci_dma_supported (struct pci_dev *dev, gulp_mask);
int pci_set_dma_mask (struct pci_dev *dev, size_t size);
void pci_alloc_consistent (struct pci_dev *dev, size_t size, dma_addr_t *da);
void pci_free_consistent (struct pci_dev *dev, size_t size, dma_addr_t *da);
void *pci_map_single (struct pci_dev *dev, void *phys, size_t size, dma_addr_t *daddr, int direction);
void *pci_unmap_single (struct pci_dev *dev, dma_addr_t daddr, 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 to set up a DMA Pool, which is essentially a slab cache intended for use in DMA transfers.

The basic functions are:

```c
#include <linux/dmaspool.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
CHAPTER 23. DIRECT MEMORY ACCESS (DMA)

23.8. DMA UNDER ISA

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:

\[
\begin{align*}
\text{dma_addr_t sg_dma_address (struct scatterlist *sg);} \\
\text{unsigned int sg_dma_len (struct scatterlist *sg);} \\
\end{align*}
\]

which return the bus (DMA) address and length of the buffer (which may be different than what was passed to \text{dma_map_sg()} because of buffer coalescence.)

After the full transfer has been made, one calls

\[
\begin{align*}
\text{int dma_unmap_sg (struct device *dev, struct scatterlist *sg, int nents,}
\text{ enum dma_data_direction direction);} \\
\end{align*}
\]

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 DMA-C (9377 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 DMA-C, 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 6377 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 DMA-C 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 PCs, there are two DMA-C'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:

\[
\begin{align*}
\text{#include <asm/dma.h>}
\text{int request_dma (unsigned int dmnr, const char *device_id);} \\
\text{void free_dma (unsigned int dmnr);} \\
\end{align*}
\]
where dmarq 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:

```c
/* in initialization */
request_irq(my_irq, my_interrupt, IRQF_DISABLE, 'my_dma', NULL);
request_dma(my_dma, 'my_dma');

/* in cleanup */
free_dma(my_dma);
free_irq(my_irq, NULL);
```

There are a number of other functions used to communicate with the DMAC, all of which are coded in arm/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 ................. 273
24.2 Datalink Layer ........................................ 276
24.3 Network Device Drivers .............................. 276
24.4 Loading/Unloading .................................... 277
24.5 Opening and Closing ................................. 278
24.6 Labs ................................................... 279

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.
24.1. NETWORK LAYERS AND DATA ENCAPSULATION

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.

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.
24.4 Loading/Unloading

Network drivers are loaded and unloaded with:

```c
#include <linux/netdevice.h>

int register_netdev (struct net_device *dev);
void unregister_netdev (struct net_device *dev);
```

Usually one registers the device in the initialization callback function and unregisters it in the exit

```
ifconfig interface [atype] 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:

```c
struct net_device *alloc_netdev (int sizesof_priv, const char *name,
                                 void (*setup)(struct net_device *));
void free_netdev (struct net_device *dev);
```

where:

- `sizesof_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 "" is chosen the name will be filled
  out dynamically as devices are brought up ("net0", "net1", ...).
- `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:

```c
struct net_device *alloc_etherdev (int sizesof_priv);
```

which supplies "" 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:
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:

```c
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 be 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:

```c
def mynet_setup(struct net_device *dev){
    ether_setup(dev);
    dev->open = mynet_open;
    dev->stop = mynet_close;
    dev->hard_start_xmit = mynet_xmit;
    dev->rx_timeout = mynet_timeout;
    dev->do_ioctl = mynet_ioctl;
    dev->get_stats = mynet_get_stats;
    dev->watchdog_timer = 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.

- Beginning with the 2.6.29 kernel many function pointers have been moved out of the net_device 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.

24.6 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:
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 .................................................. 281
25.2 `net_device_ops` Structure ............................................... 287
25.3 `sk_buff` Structure .......................................................... 289
25.4 Socket Buffer Functions .................................................. 290
25.5 Labs ............................................................................. 293

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 net_device structure elements

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>char name[IFNAMSIZ]</td>
<td>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.</td>
</tr>
<tr>
<td>unsigned long mem_end</td>
<td>Shared (on-board) memory end.</td>
</tr>
<tr>
<td>unsigned long mem_start</td>
<td>Shared memory start. (Total on-board memory = end-start.)</td>
</tr>
<tr>
<td>unsigned long base_addr</td>
<td>Device I/O address. Assigned during device probe. (0 for probing, 0x2F00 for no probing)</td>
</tr>
<tr>
<td>unsigned char irq</td>
<td>Device IRQ number</td>
</tr>
<tr>
<td>unsigned char if_port</td>
<td>On devices with multiports, specifies which port.</td>
</tr>
<tr>
<td>unsigned char dma</td>
<td>DMA channel allocated by the device (as on ISA).</td>
</tr>
<tr>
<td>unsigned long state</td>
<td>Device state, including several flags.</td>
</tr>
<tr>
<td>int features</td>
<td>Tells the kernel about any special hardware capabilities possessed by the device.</td>
</tr>
<tr>
<td>struct net_device *next</td>
<td>Next device in the linked list of devices beginning at dev_base.</td>
</tr>
<tr>
<td>int (*init)(struct net_device *dev)</td>
<td>Device initialization function.</td>
</tr>
</tbody>
</table>

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: net_device functional methods

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>int (*open)(struct net_device *dev);</td>
<td>Open the interface, whenever ifconfig activates it. Should register resources (I/O Ports, IRQ, etc.), turn on hardware, etc. (fundamental)</td>
</tr>
<tr>
<td>int (*stop)(struct net_device *dev);</td>
<td>Stop the interface. Reverse the open operations. (fundamental)</td>
</tr>
<tr>
<td>int (*hard_startmit)(struct sk_buff *skb, struct net_device *dev);</td>
<td>Hardware start transmission. Send the packet in sk_buff. (fundamental)</td>
</tr>
<tr>
<td>int (*poll)(struct net_device *dev, int *quota);</td>
<td>Method for NAPI-compliant (interrupt-mitigated) drivers to operate in poll mode, with interrupts disabled. (optional)</td>
</tr>
<tr>
<td>int (*hard_header)(struct sk_buff *skb, struct net_device *dev, unsigned short type, void *addr, void *addr, unsigned len);</td>
<td>Build the hardware header from the source and destination hardware addresses. (fundamental)</td>
</tr>
<tr>
<td>int (*rebuild_header)(struct sk_buff *skb);</td>
<td>Rebuild the hardware header before packet is transmitted. (fundamental)</td>
</tr>
<tr>
<td>void (*set_multicast_list)(struct net_device *dev);</td>
<td>Called when multicast list for the device changes, or flags are set. (optional)</td>
</tr>
<tr>
<td>int (*set_mac_address)(struct net_device *dev, void *addr);</td>
<td>If the interface permits the hardware address to be changed. (optional)</td>
</tr>
<tr>
<td>int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);</td>
<td>Perform interface-specific ioctl commands. (optional)</td>
</tr>
<tr>
<td>int (*set_config)(struct net_device *dev, struct ifmap *map);</td>
<td>Change the interface configuration. (fundamental)</td>
</tr>
<tr>
<td>struct net_device_state *(got_stat)(struct net_device *dev);</td>
<td>Gatherers statistics for reporting, such as to ifconfig. (fundamental)</td>
</tr>
<tr>
<td>struct iw_statistics *(got_wireless_stat)(struct net_device *dev);</td>
<td>Gatherers statistics for wireless devices, such as to ifconfig. (fundamental)</td>
</tr>
<tr>
<td>int (*change_mtu)(struct net_device *dev, int new_mtu);</td>
<td>If there is a change in the MTU (Maximum Transfer Unit), do actions. (optional)</td>
</tr>
<tr>
<td>void (*tx_timeout)(struct net_device *dev);</td>
<td>Handle transmission (TX) timeouts. (fundamental)</td>
</tr>
</tbody>
</table>
CHAPTER 25. NETWORK DRIVERS II: DATA STRUCTURES

25.1. NET DEVICE STRUCTURE

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned mtu;</td>
<td>Maximum transfer unit. (Default for Ethernet is 1500.)</td>
</tr>
<tr>
<td>unsigned char_type;</td>
<td>Interface hardware type. (For Ethernet is ARPHRD_ETHER.)</td>
</tr>
<tr>
<td>unsigned char addr_len;</td>
<td>MAC address length (6 for Ethernet.)</td>
</tr>
<tr>
<td>unsigned char broadcast;</td>
<td>Hardware broadcast address (6 bytes of Oxff for Ethernet.)</td>
</tr>
<tr>
<td>dev_addr [MAX_ADDR_LEN];</td>
<td>Hardware MAC address.</td>
</tr>
<tr>
<td>unsigned short flags;</td>
<td>Interface flags.</td>
</tr>
</tbody>
</table>

The flags entry is a bitmask of the following (defined in /usr/src/linux/include/linux/if.h), where IPP_ stands for interface flags:

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPP_UP</td>
<td>Interface active.</td>
</tr>
<tr>
<td>IPP_BROADCAST</td>
<td>Interface allows broadcasting.</td>
</tr>
<tr>
<td>IPP_DEBUG</td>
<td>Turn on debugging (verbose).</td>
</tr>
<tr>
<td>IPP_LOOPBACK</td>
<td>Is a loopback interface.</td>
</tr>
<tr>
<td>IPP_POINTTOPOINT</td>
<td>Indicates a point-to-point link, such as ppp.</td>
</tr>
<tr>
<td>IPP_HO trailer</td>
<td>Avoid use of trailers. For BSD compatibility only.</td>
</tr>
<tr>
<td>IPP_RUNNING</td>
<td>Interface resources allocated.</td>
</tr>
<tr>
<td>IPP_NOARP</td>
<td>Interface can't perform ARP, such as for ppp.</td>
</tr>
<tr>
<td>IPP_PROMISC</td>
<td>Interface is operating promiscuously (seeing all packets on the network.)</td>
</tr>
<tr>
<td>IPP_ALLMULTI</td>
<td>Interface should receive all multicast packets.</td>
</tr>
<tr>
<td>IPP_MASTER</td>
<td>Master interface for load equalization (balance).</td>
</tr>
<tr>
<td>IPP_SLAVE</td>
<td>Slave interface for load equalization (balance).</td>
</tr>
</tbody>
</table>

---

A number of other fields describe interface information. Some of them pointing to initializing functions are:

Table 23.3: netdevice interface information

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>void ltalk_setup</td>
<td>Initialize fields for a LocalTalk device.</td>
</tr>
<tr>
<td>(struct net_device *dev);</td>
<td></td>
</tr>
<tr>
<td>void fc_setup</td>
<td>Initialize fields for fiber channel devices.</td>
</tr>
<tr>
<td>(struct net_device *dev);</td>
<td></td>
</tr>
<tr>
<td>void fddi_setup</td>
<td>Initialize fields for fiber distributed data interface (FDDI).</td>
</tr>
<tr>
<td>(struct net_device *dev);</td>
<td></td>
</tr>
<tr>
<td>void hippi_setup</td>
<td>Initialize fields for a high-performance parallel interface (HIPPI)</td>
</tr>
<tr>
<td>(struct net_device *dev);</td>
<td>high speed interconnect driver.</td>
</tr>
<tr>
<td>void tr_config</td>
<td>Initialize fields for token ring devices.</td>
</tr>
<tr>
<td>(struct net_device *dev);</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned short</td>
<td>Hardware header length (number of bytes before the IP or</td>
</tr>
<tr>
<td>hard_header_len;</td>
<td>other protocol header, 14 for Ethernet.)</td>
</tr>
</tbody>
</table>

The features field is a bit mask of the following potential hardware capabilities of the device:

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NETIF_F_BG</td>
<td>Can use scatter/gather I/O; can transmit a packet split into distinct memory segments.</td>
</tr>
<tr>
<td>NETIF_F_IP_CSUM</td>
<td>Can do checksum of IP packets but not others.</td>
</tr>
<tr>
<td>NETIF_F_NO_CSUM</td>
<td>No checksums ever required (such as a loopback device).</td>
</tr>
<tr>
<td>NETIF_F_HW_CSUM</td>
<td>Can checksum all packets.</td>
</tr>
<tr>
<td>NETIF_F_HIGDM</td>
<td>Can DMA to high memory; otherwise all DMA is to low memory.</td>
</tr>
<tr>
<td>NETIF_F_PAGELMSY</td>
<td>Can cope with scatter/gather I/O - used in loopback.</td>
</tr>
<tr>
<td>NETIF_F_HW_VLAN_TX</td>
<td>Has transmit acceleration for 802.1q VLAN packets.</td>
</tr>
<tr>
<td>NETIF_F_HW_VLAN_RX</td>
<td>Has receive acceleration for 802.1q VLAN packets.</td>
</tr>
<tr>
<td>NETIF_F_VLAN_FILTER</td>
<td>Can receive filtering on the VLAN.</td>
</tr>
<tr>
<td>NETIF_F_VLAN2CHALLENGED</td>
<td>Gets confused and should not handle VLAN packets.</td>
</tr>
<tr>
<td>NETIF_F_TSO</td>
<td>Can perform offload TCP/IP segmentation.</td>
</tr>
<tr>
<td>NETIF_F_ILTX</td>
<td>Can do lock-less transmission.</td>
</tr>
</tbody>
</table>

25.2 net_device_ops Structure

The net_device 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 net_device structure. The detailed structure is defined in /usr/src/linux/include/linux/netdevice.h and including all conditional fields looks like:

```c
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);
    void (*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);
```

The final set of fields are utility fields and hold status information. The important ones are:

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned long last_rx;</td>
<td>The jiffies value when transmission began. (last rx is presently unused.)</td>
</tr>
<tr>
<td>int watchdog_time;</td>
<td>Minimum time (in jiffies) before the tx_timeout() function should be called.</td>
</tr>
<tr>
<td>void *priv;</td>
<td>A pointer the driver can use at will; a good place to store data.</td>
</tr>
<tr>
<td>struct dev_mc_list *mc_list;</td>
<td>Used to handle multicast transmission.</td>
</tr>
<tr>
<td>int mc_count;</td>
<td></td>
</tr>
<tr>
<td>spinlock_t xmit_lock;</td>
<td>Used to avoid multiple calls to the transmission function. (Not to be called by the driver itself.)</td>
</tr>
</tbody>
</table>
25.3 \textbf{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.

![Socket Buffer Layout](image)

Figure 25.1: Socket buffer layout

Leaving out some elements which are used only when netfilter is configured, its fields are:

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct sk_buff *next, *prev;</td>
<td>Next and previous buffers in the linked list.</td>
</tr>
<tr>
<td>struct sk_buff *list;</td>
<td>Linked list of buffers.</td>
</tr>
<tr>
<td>struct sock *sk;</td>
<td>Socket that owns the packet.</td>
</tr>
<tr>
<td>struct timeval *stamp;</td>
<td>Time the packet arrived.</td>
</tr>
<tr>
<td>struct net_device *dev;</td>
<td>Device sending or receiving this buffer.</td>
</tr>
<tr>
<td>struct net_device *real_dev;</td>
<td>Device packet arrived on.</td>
</tr>
<tr>
<td>union { ... } h;</td>
<td>Headers for the transport, network, and link layers. Can be searched for information such as source and destination addresses, etc.</td>
</tr>
<tr>
<td>union { ... } nh;</td>
<td></td>
</tr>
<tr>
<td>union { ... } mac;</td>
<td></td>
</tr>
<tr>
<td>struct dst_entry *dst;</td>
<td>Routing information.</td>
</tr>
</tbody>
</table>

The header file extensively documents each of these functional methods.

You would initialize the structure with something like:

```c
#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:

```c
struct net_device *dev;
......
#ifdef HAVE_NET_DEVICE_OPS
    dev->netdev_ops = &ndo;
#else
    dev->open = my_open;
    dev->stop = my_close;
    dev->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 2.3.10 this option will no longer be available as all in-toto 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.
25.4 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>
<thead>
<tr>
<th>Type</th>
<th>Function</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct sk_buff *</td>
<td>alloc_skb (unsigned int length);</td>
<td>Allocate a new socket buffer, give it a reference count of one, and initialize the data, tail, head pointers.</td>
</tr>
<tr>
<td>struct sk_buff *</td>
<td>dev_alloc_skb (unsigned int length);</td>
<td>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.</td>
</tr>
<tr>
<td>void</td>
<td>kfree_skb (struct sk_buff *skb);</td>
<td>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.</td>
</tr>
<tr>
<td>void</td>
<td>dev_kfree_skb (struct sk_buff *skb); dev_kfree_skb_irq (struct sk_buff *skb); dev_kfree_skb_any (struct sk_buff *skb);</td>
<td>For use in drivers. The three forms are non-interrupt context, interrupt context, or any context.</td>
</tr>
<tr>
<td>unsigned char *</td>
<td>skb_put (struct sk_buff *skb, unsigned int len);</td>
<td>Add len bytes of data to the end of the buffer, returning a pointer to the first byte of the extra data.</td>
</tr>
<tr>
<td>unsigned char *</td>
<td>skb_push (struct sk_buff *skb, unsigned int len);</td>
<td>Add len bytes of data to the beginning of the buffer, returning a pointer to the first byte of the extra data.</td>
</tr>
<tr>
<td>unsigned char *</td>
<td>skb_pull (struct sk_buff *skb, unsigned int len);</td>
<td>Remove data from the buffer start, returning a pointer to the new start of data. The space released will go into headroom.</td>
</tr>
</tbody>
</table>

char cb[48];
unsigned int len;
unsigned int data_len;
unsigned int csum;
unsigned char pkt_closed;
unsigned char pkt_type;
unsigned char ip_summed;
_u32 priority;
atomic_t users;
unsigned short protocol;
unsigned short security;
unsigned int true_size;
unsigned char *head;
unsigned char *data;
unsigned char *tail;
unsigned char *end;
void (*destructor) (struct sk_buff *);
### 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:

```c
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.

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.
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 ........................................... 295
26.2 Receiving Data ................................................................. 297
26.3 Statistics ................................................................. 297
26.4 Labs ................................................................. 298

26.1 Transmitting Data and Timeouts

Here's a simple example of a transmission function:

```c
int my_hard_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
    int len;
```
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 a 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:

```c
void netif_rx(struct net_device *dev, int len, unsigned char *buf)
{
    struct sk_buff *skb;
    skb = dev->alloc_skb(len); // Allocate a new skb
    skb_reserve(skb, 2); // Reserve space for protocol header
    memcpy(skb->buf(skb, len), buf, len); // Copy data to skb
    skb->dev = dev;
    skb->protocol = eth_type_trans(skb, dev); // Set skb protocol
    skb->ip_provided =IP_PROVIDED_HEADER; // Indicate header is provided
    skb->stats.rx_packets++; // Increment rx packet counter
    netif_rx(skb); // Call netif_rx()
}
```

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:

```c
void netif_rx(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().
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 */
collisions;

/* detailed rx_errors */
unsigned long rx_length_errors; /* receiver ring buf overflow */
unsigned long rx_over_errors; /* recvd pkt with crc error */
unsigned long rx_crc_errors; /* recvd frame alignment error */
unsigned long rx_frame_errors; /* recvd r Preston errors */
unsigned long rx_fifo_errors; /* receiver fifo overrun */
unsigned long rx_missed_errors; /* receiver missed packet */

/* detailed tx_errors */
unsigned long tx_length_errors; /* transmitter errors */
unsigned long tx_over_errors; /* transmitter errors */
unsigned long tx_fifo_errors; /* transmitter fifo errors */
unsigned long tx_carrier_errors; /* hearbeat errors */
unsigned long tx_window_errors;

/* for config 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 -Bl mynet0 192.168.3
```

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 .................................................. 302
27.2 Changes in Link State ...................................... 303
27.3 ioctl .......................................................... 303
27.4 NAPI and Interrupt Mitigation ............................ 304
27.5 NAPI Details .................................................. 304
27.6 TSO and TOE ................................................ 305
27.7 MII and ethtool .............................................. 306
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 IFP_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:

```c
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:

```c
struct dev_mc_list {
    struct dev_mc_list  *next; /* next address in list */
    struct hwaddr       *hw; /* hardware address */
    unsigned char        dmi_addresses; /* # of users */
    int                   dmi_users; /* address length */
    int                   dmi_groups; /* # of groups */
};
struct dev_mc_list *dev->mc_list;
```

There are also a number of flags (in dev->flags) which affect behaviour:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFP_MULTICAST</td>
<td>If not set, the device won’t handle multicast packets. However, the net_multicast_list() method will still be called whenever the flags change.</td>
</tr>
<tr>
<td>IFP_ALLMULTI</td>
<td>Tells the driver to retrieve all multicast packets.</td>
</tr>
<tr>
<td>IFP_PROMISC</td>
<td>Puts the interface in promiscuous mode. All packets are received regardless of what is in dev-&gt;mc_list.</td>
</tr>
</tbody>
</table>

Corbet, Rubini and Krogh-Hartman give an example of an implementation of the net_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:

```c
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 ioctl

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 SIOCSTIFADDR, 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.

```c
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 queuing 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 or on an in-memory DMA ring buffer). It should also 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 device turns off reception interrupts, and periodical 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 devices 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:

```c
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.

### 27.6 TSO and TOE

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 invoke the `poll` method to be called eventually.

The `poll` method looks like:

```c
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.

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 mbuf-sized segments 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 mbuf-sized segments on the device itself. Each fragment, or packet, is attached to 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 or 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:

```c
#include <linux/mii.h>

struct mii_if_info {
  int phy_id;
  int advertising;
  int phy_id_mask;
  int reg_max_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 *, int phy_id, int location);
  void (*mdio_write)(struct net_device *, 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 offers 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:

```c
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_vio)(struct net_device *, struct ethtool_vioinfo *);
  int (*set_vio)(struct net_device *, struct ethtool_vioinfo *);
  u32 (*get_mrgw)(struct net_device *);
  void (*get_mrgw_level)(struct net_device *, u32);
  int (*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 *);
};
```

To enable ethtool for your device you have to set a pointer to this structure in the net_device structure, using the SET_ETHTOOL_OPS macro. If MII support is also enabled, the functions mii_ethtool_get() and mii_ethtool_set() 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 involved. 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? ........................................ 310
28.2 USB Topology ........................................ 310
28.3 Descriptors .......................................... 311
28.4 USB Device Classes ................................. 312
28.5 Data Transfer ....................................... 313
28.6 USB under Linux .................................... 314
28.7 Registering USB Devices .......................... 314
28.8 Example of a USB Driver ......................... 317
28.9 Labs .................................................. 319
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 overloads 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), 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

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>
<thead>
<tr>
<th>Base Class</th>
<th>Descriptor Usage</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>Device</td>
<td>Unspecified</td>
<td>Use class information in the interface descriptors</td>
</tr>
<tr>
<td>01h</td>
<td>Interface</td>
<td>Audio</td>
<td>Speakers, microphones, sound cards</td>
</tr>
<tr>
<td>02h</td>
<td>Both</td>
<td>Communications and CDC Control</td>
<td>Network adapters, modems, serial port adapters</td>
</tr>
<tr>
<td>03h</td>
<td>Interface</td>
<td>HID (Human Interface Device)</td>
<td>Mice, joysticks, keyboards</td>
</tr>
<tr>
<td>05h</td>
<td>Interface</td>
<td>Physical</td>
<td>Force feedback joystick</td>
</tr>
<tr>
<td>06h</td>
<td>Interface</td>
<td>Image</td>
<td>Digital cameras</td>
</tr>
<tr>
<td>07h</td>
<td>Interface</td>
<td>Printer</td>
<td>Printers</td>
</tr>
<tr>
<td>08h</td>
<td>Interface</td>
<td>Mass Storage</td>
<td>Flash drives, MP3 players, memory card readers</td>
</tr>
<tr>
<td>09h</td>
<td>Device</td>
<td>Hub</td>
<td>Full and high speed hubs</td>
</tr>
<tr>
<td>0Ah</td>
<td>Interface</td>
<td>CDC-Data</td>
<td>Used together with CDC Control</td>
</tr>
<tr>
<td>0Bh</td>
<td>Interface</td>
<td>Smart Card</td>
<td>Smart card readers</td>
</tr>
<tr>
<td>0Ch</td>
<td>Interface</td>
<td>Content Security</td>
<td>Security</td>
</tr>
<tr>
<td>0Dh</td>
<td>Interface</td>
<td>Video</td>
<td>Webcams</td>
</tr>
</tbody>
</table>

### 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.h`:

```c
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 0x02
2.6.31: 82 \#define USB_REQ_GET_ADDRESS 0x03
2.6.31: 83 \#define USB_REQ_SET_ADDRESS 0x04
2.6.31: 84 \#define USB_REQ_GET_DESCRIPTOR 0x05
2.6.31: 85 \#define USB_REQ_SET_DESCRIPTOR 0x06
2.6.31: 86 \#define USB_REQ_GET_CONFIGURATION 0x07
2.6.31: 87 \#define USB_REQ_SET_CONFIGURATION 0x08
2.6.31: 88 \#define USB_REQ_GET_INTERFACE 0x09
2.6.31: 89 \#define USB_REQ_SET_INTERFACE 0x0A
2.6.31: 90 \#define USB_REQ_GET_STATUS 0x0B
2.6.31: 91 \#define USB_REQ_SET_STATUS 0x0C
2.6.31: 92 \#define USB_REQ_GET_ENCRYPTION 0x0D
2.6.31: 93 \#define USB_REQ_SET_ENCRYPTION 0x0E
2.6.31: 94 \#define USB_REQ_PIPE_RESET 0x0F
2.6.31: 95 \#define USB_REQ_PIPE_SUSPEND 0x0F
2.6.31: 96 \#define USB_REQ_PIPE_RESUME 0x10
2.6.31: 97 \#define USB_REQ_GET_PIPE 0x11
2.6.31: 98 \#define USB_REQ_SET_PIPE 0x12
2.6.31: 99 \#define USB_REQ_GET_SECURITY_DATA 0x13
2.6.31: 100 \#define USB_REQ_SET_SECURITY_DATA 0x14
2.6.31: 101 \#define USB_REQ_GET_SECURITY_DATA 0x15
2.6.31: 102 \#define USB_REQ_SET_SECURITY_DATA 0x16
2.6.31: 103 \#define USB_REQ_GET_INTERFACE_ID 0x17
```

Other devices require a fully customized device driver be written.
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.

![USB Stack Diagram](image)

Figure 28.3: USB: Controller, Core and Device

The USB core has APIs 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:

```c
#include <linux/usb.h>

int usb_register (struct usb_device *dev);
void usb_deregister (struct usb_device *dev);

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:

```
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:

```c
2.6.31: 432 static struct
2.6.31: 433 file_operations usb_rrio_fops = {          
2.6.31: 434 .owner = THIS_MODULE,
2.6.31: 435 .read = read_rrio,
2.6.31: 436 .write = write_rrio,
2.6.31: 437 .unlocked_ioctl = ioct1_rrio,
2.6.31: 438 .open = open_rrio,
2.6.31: 439 .release = close_rrio,
2.6.31: 440 );
2.6.31: 441
2.6.31: 442 static struct usb_class_driver usb_rrio_class = {  
2.6.31: 443 .name = "rio500",  
2.6.31: 444 .fops = &usb_rrio_fops,
2.6.31: 445 .module = 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:

```c
2.6.31: 514 static struct usb_device_id rrio_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, rrio_table);
2.6.31: 520
2.6.31: 521 static struct usb_driver rrio_driver = {  
2.6.31: 522 .name = "rio500",
2.6.31: 523 .probe = probe_rrio,
2.6.31: 524 .disconnect = disconnect_rrio,
2.6.31: 525 };
```
Note that the init function simply registers the usb_driver structure:

```c
2.6.31: 528 static int __init usb_rio_init(void)
2.6.31: 529 {
2.6.31: 530    int retval;
2.6.31: 531    retval = usb_register(rio_driver);
2.6.31: 532    if (retval)
2.6.31: 533      goto out;
2.6.31: 534    printk(KERN_INFO "RIO_DRIVER\n\"");
2.6.31: 535    return retval;
2.6.31: 540 }
```

Likewise, the cleanup, or exit, function simply unregisters:

```c
2.6.31: 543 static void __exit usb_rio_cleanup(void)
2.6.31: 544 {
2.6.31: 545    struct rio_usb_data *rio = &rio_instance;
2.6.31: 546    rio->present = 0;
2.6.31: 547    usb_deregister(rio_driver);
2.6.31: 549 }
```

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:

```c
2.6.31: 448 static int probe_rio(struct usb_interface *intf,
2.6.31: 449    const struct usb_device_id *id)
2.6.31: 450 {
2.6.31: 451    struct usb_device *dev = interface_to_usbdev(intf);
2.6.31: 452    struct rio_usb_data *rio = &rio_instance;
2.6.31: 453    int retval;
2.6.31: 454    dev_info(intf->dev_info, "USB Rio found at address %d\n", dev->devnum);
2.6.31: 455    retval = usb_register_dev(intf, &rio clase);
2.6.31: 456    if (retval) {
2.6.31: 457      err("Not able to get a minor for this device.\n");
2.6.31: 460      return -ENOMEM;
2.6.31: 461    }
2.6.31: 462    rio->rio_dev = dev;
2.6.31: 463 }
```

2.9. 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 printk() 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? .................. 321
29.2 NAND vs. NOR .............................. 322
29.3 Driver and User Modules .................. 324
29.4 Flash Filesystems ............................ 324
29.5 Labs ...................................... 325

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

<table>
<thead>
<tr>
<th>Link</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.linux-mtd.infradead.org">http://www.linux-mtd.infradead.org</a></td>
<td>The main web site for Linux MTD development.</td>
</tr>
<tr>
<td><a href="http://www.linuxdevices.com/articles/YT7478621147.html">http://www.linuxdevices.com/articles/YT7478621147.html</a></td>
<td>A white paper by Cliff Brake and Jeff Sutherland about using flash in embedded Linux systems.</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Interface</th>
<th>NOR</th>
<th>NAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus</td>
<td>Large</td>
<td>Small</td>
</tr>
<tr>
<td>Cell Size</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Read Time</td>
<td>Fast</td>
<td>Slow</td>
</tr>
<tr>
<td>Program Time (single byte)</td>
<td>Fast</td>
<td>Slow</td>
</tr>
<tr>
<td>Program Time (multi byte)</td>
<td>Slow</td>
<td>Fast</td>
</tr>
<tr>
<td>Erase Time</td>
<td>Slow</td>
<td>Fast</td>
</tr>
<tr>
<td>Power consumption</td>
<td>High</td>
<td>Low, but requires additional RAM</td>
</tr>
<tr>
<td>Can execute code</td>
<td>Yes</td>
<td>No, but newer chips can execute a small loader out of the first page</td>
</tr>
<tr>
<td>Bit twiddling</td>
<td>nearly unrestricted</td>
<td>I-3 times, also known as “partial page program restriction”</td>
</tr>
<tr>
<td>Bad blocks at ship time</td>
<td>No</td>
<td>Allowed</td>
</tr>
</tbody>
</table>
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.

29.4 Flash Filesystems

Filesystems for flash devices pose some important challenges:

- Block sizes can be relatively large (94 KB to 256 KB). Under present Linux implementations, a block device filesystem cannot 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 be locked out.
- Flash memory is expensive, so compressed filesystems are attractive.
- Journaled 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 turn.

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.

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 JFFS2(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:
Before or after this, you’ll have to make sure to do

```bash
modprobe addrmap total_size=2048 erase_size=6
```

(or leave 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 an 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 `mks_jffs2` utility, which you can download in source or binary form from [http://sources.redhat.com/jffs2](http://sources.redhat.com/jffs2).

You’ll need to do

```bash
modprobe mdblock
```

if you haven’t built this into the kernel.

You’ll also have to make the proper device node:

```bash
mkdev -m 666 /dev/mdblock0 b 31 0
```

Populate a directory tree (say `.dir_tree`) with some files and subdirectories; the total size should be less than 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:

```bash
mks_jffs2 -d .dir_tree -o /dev/mdblock0
```

(You may want to separate out these steps so you can keep the initial filesystem image; i.e., do something like

```bash
mks_jffs2 -d .dir_tree -o jfs.image
dd if=jfs.image of=/dev/mdblock0
```

Now you can mount the filesystem and play with it to your heart’s content:

```bash
mkdir .mnt_jffs2
mount -t jffs2 /dev/mdblock0 .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
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 ........................................ 329
30.2 APM and ACPI ............................................. 330
30.3 System Power States ........................................ 331
30.4 Callback Functions ........................................ 332
30.5 Labs ....................................................... 334

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.
CHAPTER 30. POWER MANAGEMENT

At a lower level the device_driver structure will then contain pointers to these functions:

```c
struct device_driver {
    int (*probe) (struct device *dev);
    int (*remove) (struct device *dev);
    void (*suspend) (struct device *dev);
    int (*pm_enable) (struct device *dev, pm_message_t state);
    void (*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, documenta-
tion, 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 compli-
cated.

30.3 SYSTEM POWER STATES

Under APM there are five states the system can be in:

<table>
<thead>
<tr>
<th>State</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full On</td>
<td>Default mode. No power management. All devices are on.</td>
</tr>
<tr>
<td>APM Enabled</td>
<td>System does work; some unused devices may not be powered. CPU clock may be slowed or stopped.</td>
</tr>
<tr>
<td>APM Standby</td>
<td>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.</td>
</tr>
<tr>
<td>APM Suspend</td>
<td>Enters this state after long period of inactivity; recovery 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.</td>
</tr>
<tr>
<td>Off</td>
<td>System off. All power off. Nothing saved.</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>State</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mechanical Off</td>
<td>G3</td>
<td>Power consumption is zero except for the real-time clock.</td>
</tr>
<tr>
<td>Soft Off</td>
<td>G2/S5</td>
<td>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.</td>
</tr>
<tr>
<td>Sleeping</td>
<td>G1</td>
<td>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.</td>
</tr>
</tbody>
</table>
CHAPTER 30. POWER MANAGEMENT

<table>
<thead>
<tr>
<th>Working</th>
<th>G0</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Volatile Sleep</td>
<td>S4</td>
<td>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.</td>
</tr>
</tbody>
</table>

Under ACPI there are also a bunch of Device Power State Definitions. They range from D0 to D3. All devices must have D0 and D3 modes defined, but D1 and D2 are optional.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>D3</td>
<td>Off. No power to the device. Context is lost; operating system must re-initialize upon re-powering.</td>
</tr>
<tr>
<td>D2</td>
<td>Meaning depends on the class of device. Saves more power than D1. Device may lose some of its context.</td>
</tr>
<tr>
<td>D1</td>
<td>Meaning depends on the class of the device. Saves more context than D2.</td>
</tr>
<tr>
<td>D0</td>
<td>Full power to the device, which is completely active, retaining all context.</td>
</tr>
</tbody>
</table>

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:

```c
struct pci_driver {
    struct list_head node;
    char *name;
    ...
};
```

```c
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);
```

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:

```c
struct usb_driver {
    struct module *owner;
    const char *name;
    int (*probe)(struct usb_interface *int, const struct usb_device_id *id);
    void (*disconnect)(struct usb_interface *int);
    int (*event)(struct usb_interface *int, unsigned int code, void *buf);
    int (*suspend)(struct usb_interface *int, pm_message_t message);
    int (*resume)(struct usb_interface *int);
    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

```c
#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 req,
                    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 geostatic dependencies, etc., 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 excellent description of the old and new interfaces look at http://tree.celinuxforum.org/CeliPubWiki/PmSubSystem.

30.5 Labs

Lab 1: Monitoring Power Usage with powertop

A tool for assessing power consumption can be obtained from http://www.lewiswatts.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 preexisting one. We'll explain how to write callback functions and insert them in the relevant chain.

31.1 What are Notifiers? ........................................ 335
31.2 Data Structures ........................................ 336
31.3 Callbacks and Notifications .............................. 337
31.4 Creating Notifier Chains ................................. 337
31.5 Labs ....................................................... 338

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:
31.3. CALLBACKS AND NOTIFICATIONS

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 NOTIFYER_STOP_MARKER is set in the callback function return value, any notifier can stop any further processing. Other return values are not confused, 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:

```c
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:

```c
int blocking_notifier_call_chain (struct blocking_notifier_head *nh,
                                 unsigned long event, void *data);
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:

```c
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 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 callback 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 kinds 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_NOTIFYER or CPUFREQ_POLICY_NOTIFYER. 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_freqd {
2.6.31: 124     unsigned int cpu;
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 event has the value CPUFREQ_COMPATIBLE. Changes should only be made to avoid hardware failure.

The third 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: 62 struct cpufreq_policy {
2.6.31: 63     cpumask_var_t cpus;    /* CPUs requiring no coordination */
2.6.31: 64     cpumask_var_t related_cpus; /* CPUs with any coordination */
2.6.31: 65     unsigned int shared_type; /* any or all affected CPUs */
2.6.31: 66     unsigned int should_not_cpufreq; /* governors are used */
2.6.31: 67     unsigned int cpu;         /* cpu nr of registered CPU */
2.6.31: 68     struct cpufreq_cpubinfo cpubinfo; /* see above */
2.6.31: 69     unsigned int min;         /* in kHz */
2.6.31: 70     unsigned int max;         /* in kHz */
2.6.31: 71     unsigned int cur;         /* in kHz, only needed if cpufreq
2.6.31: 72     unsigned int policy;      /* see above */
2.6.31: 73     struct cpufreq_governor *governor; /* see below */
2.6.31: 74     unsigned int update;      /* if update_policy() needs to be
2.6.31: 75     struct cpuinfo_uk *uk;        /* called, but you’re in UK context */
```

```
CHAPTER 32. CPU FREQUENCY SCALING

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 always 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:

```c
#include <linux/cpufreq.h>

int cpufreq_register_driver (struct cpufreq_driver *driver_data);
void cpufreq_unregister_driver (struct cpufreq_driver *driver_data);
```

where the cpufreq_driver structure should be filled out first, and looks like:

```c
struct cpufreq_driver {
  struct module *owner;
  char name[CPufreq_NAME_LEN];
  char *flags;
  int (*init) (struct cpufreq_policy *policy);
  int (*void_init) (struct cpufreq_policy *policy);
  int (*acquire) (struct cpufreq_policy *policy);
  int (*release) (struct cpufreq_policy *policy);
  int (*set>m)
```

The int() 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.

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:

```c
#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:

```c
2.6.31: 165 struct cpufreq_governor {
  struct module *owner;
  char name[CPufreq_NAME_LEN];
  int (*set_policy) (struct cpufreq_policy *policy, 
                    unsigned int target_freq);
  int (*set_freq) (struct cpufreq_policy *policy, 
                   unsigned int freq);
  int (*set_flags) (struct cpufreq_policy *policy, 
                    unsigned int flags);
  int (*set_frequency) (struct cpufreq_policy *policy, 
                        unsigned int freq);
  struct freq_attr **attr;
};
```

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:

- CPURRQ_GOV_START: Start operating for this CPU.
- CPURRQ_GOV_STOP: End operating for this CPU.
CHAPTER 32. CPU FREQUENCY SCALING

- CPUFREQ_LIMITS: The maximum and minimum limits have changed for this CPU.

The callback function can call the CPU frequency driver using:

```c
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_R (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? ........................................ 345
33.2 The Posix Asynchronous I/O API ................................ 346
33.3 Linux Implementation ........................................... 347
33.4 Labs ................................................................. 350

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 inter-
action 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.
33.2 The Posix Asynchronous I/O API

The POSIX 1b standard defines a basic data structure, the aioch, 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:

```c
struct aioch {
    int aio_fileno; /* File descriptor. */
    int aio_lio_opcode; /* Operation to be performed. */
    int aio_recprio; /* 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_fileno may 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_recprio 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, llbri. These functions are:

```c
#include <aio.h>

void aio_init (const struct aioinit *init);
int aio_read (struct aioch *cb);
```

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 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/1o/aionotes.txt](http://lse.sourceforge.net/1o/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:

```c
seize_t (*aio_read) (struct kiocb *iocb, const struct iovec *iovec, unsigned long iov, loff_t pos);
```
CHAPTER 33. ASYNCHRONOUS I/O

 seins_t (*aio_write)(struct ioknob *iocb, const struct iovec *iov, unsigned long niov, loff_t pos);
 int (*aio_fsync)(struct ioknob *, 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, long nr, struct ioch *iobpp);
long io_getevents (aio_context_t ctx, long nr, long nmr, long nr, struct io_event *events, struct timespec *timeout);
long io_destroy (aio_context_t ctx);
long io_cancel (aio_context_t ctx, struct iocb *iob, 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_regvrio;
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 `-lai0` 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 ................................................................. 351
34.2 Tunables ............................................................................... 353
34.3 noop I/O Scheduler .............................................................. 353
34.4 Deadline I/O Scheduler ....................................................... 354
34.5 Completely Fair Queue Scheduler ....................................... 355
34.6 Anticipatory I/O Scheduler .................................................. 355
34.7 Labs ...................................................................................... 356

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 phy-
CHAPTER 34. I/O SCHEDULING

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/ada/queue/scheduler
noop [anticipatory] deadline cfq
$ echo cfq > /sys/block/ada/queue/scheduler
$ cat /sys/block/ada/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/iostats

For example:

$ ls -l /sys/block/ada/queue/iostats
total 0
-rw-r--r-- 1 root root 4096 Mar 27 17:42 anticipatory
-rw-r--r-- 1 root root 4096 Mar 27 17:42 deadline
-rw-r--r-- 1 root root 4096 Mar 27 17:42 cfq

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.

Dequeueing 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)

queue
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 served immediately.

The algorithm is adaptive in that it 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 = \( \text{HZ}/150 = 6 \))

*read_expire:*

How long (in milliseconds) until a read request expires, as well as the interval between serving expired
requests. (Default = \( \text{HZ}/8 = 125 \))

*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
= \( \text{HZ}/2 = 500 \))

*write_expire and write_batch_expire:*

Serve the same functions for writes. (Defaults = \( \text{HZ}/4 = 250 \), \( \text{HZ}/8 = 125 \))

*est_time:*

Gives some statistics.

Some detailed documentation can be found at: /usr/src/linux/Documentation/block/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.
CHAPTER 35. BLOCK DRIVERS

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 gemdsk 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 routines, just as for a character device. The functions for doing this are:

```c
#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=256.

`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 gemdsk data structure, as we will show shortly.

The `block_device_operations` structure is defined in `/usr/src/linux/include/linux/fs.h` as:

```c
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 **baddr, unsigned long *paddr);
    int (media_changed) (struct gemdsk *gd);
    int (invalidate_cache) (struct gemdsk *gd);
    int (setgeometry)(struct block_device *bdev, struct hdGeometry *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:

```c
static struct block_device_operations mybdrv_fops = {
    .owner = THIS_MODULE,
    .open = mybdrv_open,
    .release = mybdrv_release,
    .ioct1 = mybdrv_ioctl
};
```

- The 2.6.28 kernel introduces changes to the block_device_operations structure which is now defined as:

```c
struct block_device_operations {
    int (open) (struct block_device *bdev, fnode_t mode);
    int (release) (struct gendisk *gd, fnode_t mode);
    int (*locked_ioctl) (struct block_device *bdev, fnode_t mode,
                         unsigned cmd, unsigned long arg);
    int (ioctl) (struct block_device *bdev, fnode_t mode,
                  unsigned cmd, unsigned long arg);
    long (*request_ioctl) (struct block_device *bdev, fnode_t mode,
                            unsigned cmd, unsigned long arg);
    int (*direct_access) (struct block_device *bdev, fnode_t mode,
                          sector_t sector, void **skaddr,
                          unsigned long *rpm);
    int (*media_changed) (struct gendisk *gd);
    int (*validate_disk) (struct gendisk *gd);
    int (*getgeo) (struct block_device *bdev, struct hd_geometry *geo);
    struct module *module;
};
```

35.4 gendisk Structure

The gendisk structure is defined in /usr/src/linux/include/linux/genhd.h and describes a partitionable device. You'll have to set it up, manipulate it, and free it when done.

The gendisk structure is:

```c
struct gendisk {
    int major;
    /* major number of driver */
    int first_minor;
};
```

```c
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 *spart; /* [indexed by minor] */
int part_weapon_suppress;
struct block_device_operations *fops;
struct request_queue *queue;
void *private_data;
sector_t capacity;

int flags;
struct device *driverfs_dev;
struct kobject *obj;
struct kobject *holder_dir;
struct kobject *slave_dir;

struct timer_random_state *random;
int policy;

atomic_t sync_io;     /* RAID */
unsigned long stamp;
int in_flight;

#define CHIFSC_SIG    
#define disk_state diskstats;
#define lee disk_state diskstats;
#define 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 associates 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 mimicked 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:

```c
#include <linux/genhd.h>

struct gendisk *alloc_disk (int minors);
void add_disk (struct gendisk *disk);
void put_disk (struct gendisk *disk);
```
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/bkdev.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/bkdev.h>

struct request_queue *blk_init_queue (request_func *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:

```
35.5. REQUEST HANDLING

static void my_request (struct request_queue *q){
    struct request *rq;
    int size;
    char *ptr;
    printk(KERN_INFO "entering request routine\n");
    while ((rq = q->next_request (rq))){
        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 routine\n");
}
```

Pooling off the first request from the queue is done with blk_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.)
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 264 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 device should be made automatically when you load the driver; otherwise you will have to actually add the node with the insmod 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/mybdv`

`mount /dev/mybdv 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 HDRG_GETGEO, and you’ll have to include: linux/hdreg.h. This ioctl returns a pointer to the following structure:

```c
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

- get_free_page(), 188
- get_free_pages(), 188

access.ok(), 196
ACPI, 330
add_timer(), 129
AGP, 261
aio.error(), 346
aio.init(), 346
aio.read(), 346
aio.write(), 346
atohc, 346
APM, 330
asynchronous I/O, 345
atomic variables, 139
atomic functions, 139
atomic operations, 138

big kernel lock (BKL), 143
binary blobs, 14
bit operations, 138
bit functions, 140
block drivers
  blk.register_region(), 359
  blk.unregister_region(), 359

block drivers
  blk.cleanup_queue(), 362
  blk.init_queue(), 362
  blk.queue_hardsect_size(), 362
  blk.queue_logical_block_size(), 364
  block_device_operations, 359
gendisk(), 361
gendev(), 358
get_pages(), 362
guest(), 358

request function, 358, 362
request queue, 361, 362

bootums, 189
BUG(), 114
BUG_ON(), 114
buses, 240

callback functions, 16
character drivers
  file_operations, 44
  registration/do-registration, 41
character devices, 12
character drivers, 36
access, 40
cdev.alloc(), 41
cdev.del(), 41
cdev, 41
dynamical allocation, 39
driver, 40
inode, 50
major and minor numbers, 36
reserving major and minor numbers, 38
system calls, 40
usage count, 51

checkpatch.pl, 79
completion functions, 139, 148
container_of(), 174, 232
converting time values, 126
copy_from_user(), 196
copy_to_user(), 196

crash, 117
current, 70, 200
debufbe, 118
DECLARE_MUTEX(), 146
DECLARE_BSS(), 146
defetable functions, 227
deferred task, 16
<table>
<thead>
<tr>
<th>MODULE DEVICE TABLE (33)</th>
</tr>
</thead>
<tbody>
<tr>
<td>module exit (24, 107)</td>
</tr>
<tr>
<td>module init (24, 107)</td>
</tr>
<tr>
<td>module librc (24)</td>
</tr>
<tr>
<td>module module (104)</td>
</tr>
<tr>
<td>module parame (27)</td>
</tr>
<tr>
<td>parameters, 27</td>
</tr>
<tr>
<td>request module (106)</td>
</tr>
<tr>
<td>rmmod, 26</td>
</tr>
<tr>
<td>tainted, 24</td>
</tr>
<tr>
<td>mlock (128)</td>
</tr>
<tr>
<td>mlock interruptible (128)</td>
</tr>
<tr>
<td>mtd (32)</td>
</tr>
<tr>
<td>cramfs, 325</td>
</tr>
<tr>
<td>filesystems, 324</td>
</tr>
<tr>
<td>init, 324</td>
</tr>
<tr>
<td>jfs, 325</td>
</tr>
<tr>
<td>jbd, 325</td>
</tr>
<tr>
<td>NAND, 322</td>
</tr>
<tr>
<td>NOR, 322</td>
</tr>
<tr>
<td>ramfs, 325</td>
</tr>
<tr>
<td>raw block modules, 324</td>
</tr>
<tr>
<td>raw character modules, 324</td>
</tr>
<tr>
<td>mutex init (145)</td>
</tr>
<tr>
<td>mutex lock (145)</td>
</tr>
<tr>
<td>mutex lock interruptible (144)</td>
</tr>
<tr>
<td>mutex lock (killable, 144)</td>
</tr>
<tr>
<td>mutex retry lock (145)</td>
</tr>
<tr>
<td>mutex unlock (144)</td>
</tr>
<tr>
<td>mutexes, 139, 144, 147</td>
</tr>
<tr>
<td>ndiswrapper, 15</td>
</tr>
<tr>
<td>network devices</td>
</tr>
<tr>
<td>net device ops, 278</td>
</tr>
<tr>
<td>network devices</td>
</tr>
<tr>
<td>alloc etherdev (277)</td>
</tr>
<tr>
<td>alloc netdev (277)</td>
</tr>
<tr>
<td>clcs (278)</td>
</tr>
<tr>
<td>ether_setup (277)</td>
</tr>
<tr>
<td>else (277)</td>
</tr>
<tr>
<td>net device ops, 287</td>
</tr>
<tr>
<td>netdev priv (277)</td>
</tr>
<tr>
<td>netif carrier off (303)</td>
</tr>
<tr>
<td>netif carrier ok (303)</td>
</tr>
<tr>
<td>netif carrier on (303)</td>
</tr>
<tr>
<td>netif start queue (279)</td>
</tr>
<tr>
<td>netif stop queue (279)</td>
</tr>
<tr>
<td>netif_make queue (279)</td>
</tr>
<tr>
<td>polling, 304</td>
</tr>
</tbody>
</table>

| PAGE OFFSET (184)        |
| pages, 186               |
| panic (114)              |
| patches, 78              |
| PCI, 253                 |
| BIOS, 255                |
| configuration register, 254, 258|
| configuration registers, 259|
| device detection, 255    |
| I/O resources, 260       |
| registering drivers, 256 |
| PCI express, 94, 261     |
| pci dev (174, 258)       |
| PCI device (33, 257)     |
| pci device id (237)      |
| pci driver (174, 256)    |
| pci get device (268)     |
| pci name (259)           |
| pci register device (174)|
| pci register driver (256)|
| pci unregister device (174)|
| pci unregister driver (256)|
| pid (70)                 |
| pid task (170)           |
| polling, 220             |
| power management (16, 329)|
| power management functions, 332|
| power states (331)       |
| printk (19)              |
| priority inversion (228) |
| proc filesystem         |
| seqfile interface, 165   |
| proc filesystem, 161     |
| creating entries 162     |
| reading from, 163        |
| writing to, 164          |
| proc dir entry, 162      |
| put device (173)         |
| put driver (174)         |
| put user (186)           |
| raw I/O, 196             |
| RCU, 139                 |
| read lock irqstore, 142  |
| read lock irqsave, 142   |
| reference counting, 149  |
| register jprobe (121)    |
| register jprobe (120)    |
| register netdev (277)    |
| relay channels, 267, 208 |
| release firmware, 180    |
| remorse pfn range, 206   |
| request firmware (180)   |
| rw lock init (142)       |
schedule(), 70
SCSI devices, 13
sema_init(), 146
semaphores, 96, 139, 145
segslocks, 139
setpci, 260
SLAB, 192
slab allocator, 190
sleeping, 96, 128, 213, 214, 216
sleeping, exclusive, 218
SLUB, 192
socket buffers, 275
sockets, 275
softirqs, 129, 227
spare, 70
spin_is_locked(), 142
spin_lock(), 141
spin_lock_init(), 141
spin_lock_irqsave(), 141
spin_lock.restore(), 141
spin_trylock(), 142
spin_unlock(), 141
spin_unlock_wait(), 142
spinlocks, 95, 138, 141
strace, 72
sync, 588
sysfs, 118, 171, 175, 255
syslogd, 20
system calls, 69
system tap, 122

wait, 15, 179
TASK_INTERRUPTIBLE, 214
TASK_KILLABLE, 214
TASK_RUNNING, 214
task_struct, 70, 72, 214
TASK_UNINTERRUPTIBLE, 214
taskstats, 226-228
tgid, 70
time stamp counter (TSC), 127
timer_list, 129
to_pci_dev(), 174

udhc, 42, 60, 107
udhc.conf, 43
unified device model, 171
unlikely(), 80
unlock kernel(), 143
unregister_jprobe(), 121
unregister_jprobe(), 120
unregister_netdev(), 277

up(), 145
up_read(), 145
up_write(), 145
USB, 310
bulk transfers, 314
configurations, 311
controllers, 311
descriptors, 311
device classes, 312
EHCI, 311
endpoints, 312
interrupt transfers, 314
isochronous transfers, 314
OHCI, 311
registering devices, 314
speed, 310
topology, 310
transfer types, 313
UHCI, 311
usb_deregister(), 314
USB_DEVICE(), 316
usb_device_id(), 315
usb_driver(), 314
usb_register(), 314
USB devices, 15
user address, 196
user-space, 196
user-space drivers, 14

vfree(), 189
virtual address, 184
virtual memory, 184
virtualization, 69
vm_area_struct, 204
vmalloc(), 189

wait queues, 213
wait_event(), 214
wait_event_interruptible(), 214
wait_event_timeout(), 214
wait_queue_head_t, 214
wake_up(), 214
wake_up_interruptible(), 214
waking up, 214, 218
work queues, 227
work queues, 227, 231
write_lock_irqrestore(), 142
write_lock_irqsave(), 142