Chapter 3. Preliminaries

**3.1. Modules vs Programs**

**3.1.1. How modules begin and end**

A program usually begins with a main() function, executes a bunch of instructions and terminates upon completion of those instructions. Kernel modules work a bit differently. A module always begin with either the init\_module or the function you specify with module\_init call. This is the entry function for modules; it tells the kernel what functionality the module provides and sets up the kernel to run the module's functions when they're needed. Once it does this, entry function returns and the module does nothing until the kernel wants to do something with the code that the module provides. All modules end by calling either cleanup\_module or the function you specify with the module\_exit call. This is the exit function for modules; it undoes whatever entry function did. It unregisters the functionality that the entry function registered. Every module must have an entry function and an exit function. Since there's more than one way to specify entry and exit functions, I'll try my best to use the terms `entry function' and `exit function', but if I slip and simply refer to them as init\_module and cleanup\_module, I think you'll know what I mean.

**3.1.2. Functions available to modules**

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

example) is available, and fixes the call instruction to point to that code. Kernel modules are different here, too. In the hello world example, you might have noticed that we used a function, printk() but didn't include a standard I/O library. That's because modules are object files whose symbols get resolved upon insmod'ing. The definition for the symbols comes from the kernel itself; the only external functions you can use are the ones provided by the kernel. If you're curious about what symbols have been exported by your kernel, take a look at /proc/kallsyms. One point to keep in mind is the difference between library functions and system calls. Library functions are higher level, run completely in user space and provide a more convenient interface for the programmer to the functions that do the real work−−−system calls. System calls run in kernel mode on the user's behalf and are provided by the kernel itself. The library function printf() may look like a very general printing function, but all it really does is format the data into strings and write the string data using the low−level system call write(), which then sends the data to standard output.

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

*#include <stdio.h>*

*int main(void)*

*{ printf("hello"); return 0; }*

with gcc −Wall −o hello hello.c. Run the exectable with strace ./hello. Are you impressed? Every line you see corresponds to a system call. strace[4] is a handy program that gives you details about what system calls a program is making, including which call is made, what its arguments are what it returns. It's an invaluable tool for figuring out things like what files a program is trying to access. Towards the end, you'll see a line which looks like write(1, "hello", 5hello). There it is. The face behind the printf() mask. You may not be familiar with write, since most people use library functions for file I/O (like fopen, fputs, fclose). If that's the case, try looking at man 2 write. The 2nd man section is devoted to system calls (like kill() and read(). The 3rd man section is devoted to library calls, which you would probably be more familiar with (like cosh() and random()). You can even write modules to replace the kernel's system calls, which we'll do shortly. Crackers often make use of this sort of thing for backdoors or trojans, but you can write your own modules to do more benign things, like have the kernel write Tee hee, that tickles! everytime someone tries to delete a file on your system.

**3.1.3. User Space vs Kernel Space**

A kernel is all about access to resources, whether the resource in question happens to be a video card, a hard drive or even memory. Programs often compete for the same resource. As I just saved this document, updatedb started updating the locate database. My vim session and updatedb are both using the hard drive concurrently. The kernel needs to keep things orderly, and not give users access to resources whenever they feel like it. To this end, a CPU can run in different modes. Each mode gives a different level of freedom to do what you want on the system. The Intel 80386 architecture has 4 of these modes, which are called rings. Unix uses only two rings; the highest ring (ring 0, also known as `supervisor mode' where everything is allowed to happen) and the lowest ring, which is called `user mode'. Recall the discussion about library functions vs system calls. Typically, you use a library function in user mode. The library function calls one or more system calls, and these system calls execute on the library function's behalf, but do so in supervisor mode since they are part of the kernel itself. Once the system call completes its task, it returns and execution gets transfered back to user mode.

**3.1.4. Name Space**

When you write a small C program, you use variables which are convenient and make sense to the reader. If, on the other hand, you're writing routines which will be part of a bigger problem, any global variables you have are part of a community of other peoples' global variables; some of the variable names can clash. When a program has lots of global variables which aren't meaningful enough to be distinguished, you get namespace pollution. In large projects, effort must be made to remember reserved names, and to find ways to develop a scheme for naming unique variable names and symbols. When writing kernel code, even the smallest module will be linked against the entire kernel, so this is definitely an issue. The best way to deal with this is to declare all your variables as static and to use a well−defined prefix for your symbols. By convention, all kernel prefixes are lowercase. If you don't want to declare everything as static, another option is to declare a symbol table and register it with a kernel. We'll get to this later. The file /proc/kallsyms holds all the symbols that the kernel knows about and which are therefore accessible to your modules since they share the kernel's codespace.

**3.1.5. Code space**

Memory management is a very complicated subject−−−the majority of O'Reilly's `Understanding The Linux Kernel' is just on memory management! We're not setting out to be experts on memory managements, but we do need to know a couple of facts to even begin worrying about writing real modules. If you haven't thought about what a segfault really means, you may be surprised to hear that pointers don't actually point to memory locations. Not real ones, anyway. When a process is created, the kernel sets aside a portion of real physical memory and hands it to the process to use for its executing code, variables, stack, heap and other things which a computer scientist would know about[5]. This memory begins with 0x00000000 and extends up to whatever it needs to be. Since the memory space for any two processes don't overlap, every process that can access a memory address, say 0xbffff978, would be accessing a different location in real physical memory! The processes would be accessing an index named 0xbffff978 which points to some kind of offset into the region of memory set aside for that particular process. For the most part, a process like our Hello, World program can't access the space of another process, although there are ways which we'll talk about later. The kernel has its own space of memory as well. Since a module is code which can be dynamically inserted and removed in the kernel (as opposed to a semi−autonomous object), it shares the kernel's codespace rather than having its own. Therefore, if your module segfaults, the kernel segfaults. And if you start writing over data because of an off−by−one error, then you're trampling on kernel data (or code). This is even worse than it sounds, so try your best to be careful. By the way, I would like to point out that the above discussion is true for any operating system which uses a monolithic kernel[6]. There are things called microkernels which have modules which get their own codespace. The GNU Hurd and QNX Neutrino are two examples of a microkernel.

**3.1.6. Device Drivers**

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

**3.1.6.1. Major and Minor Numbers**

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

*# ls −l /dev/hda[1−3]*

*brw−rw−−−− 1 root disk 3, 1 Jul 5 2000 /dev/hda1*

*brw−rw−−−− 1 root disk 3, 2 Jul 5 2000 /dev/hda2*

*brw−rw−−−− 1 root disk 3, 3 Jul 5 2000 /dev/hda3*

Notice the column of numbers separated by a comma? The first number is called the device's major number. The second number is the minor number. The major number tells you which driver is used to access the hardware. Each driver is assigned a unique major number; all device files with the same major number are controlled by the same driver. All the above major numbers are 3, because they're all controlled by the same driver. The minor number is used by the driver to distinguish between the various hardware it controls. Returning to the example above, although all three devices are handled by the same driver they have unique minor numbers because the driver sees them as being different pieces of hardware. Devices are divided into two types: character devices and block devices. The difference is that block devices have a buffer for requests, so they can choose the best order in which to respond to the requests. This is important in the case of storage devices, where it's faster to read or write sectors which are close to each other, rather than those which are further apart. Another difference is that block devices can only accept input and return output in blocks (whose size can vary according to the device), whereas character devices are allowed to use as many or as few bytes as they like. Most devices in the world are character, because they don't need this type of buffering, and they don't operate with a fixed block size. You can tell whether a device file is for a block device or a character device by looking at the first character in the output of ls −l. If it's `b' then it's a block device, and if it's `c' then it's a character device. The devices you see above are block devices. Here are

some character devices (the serial ports):

*crw−rw−−−− 1 root dial 4, 64 Feb 18 23:34 /dev/ttyS0*

*crw−r−−−−− 1 root dial 4, 65 Nov 17 10:26 /dev/ttyS1*

*crw−rw−−−− 1 root dial 4, 66 Jul 5 2000 /dev/ttyS2*

*crw−rw−−−− 1 root dial 4, 67 Jul 5 2000 /dev/ttyS3*

If you want to see which major numbers have been assigned, you can look at /usr/src/linux/Documentation/devices.txt. When the system was installed, all of those device files were created by the mknod command. To create a new char device named `coffee' with major/minor number 12 and 2, simply do mknod /dev/coffee c 12 2.

You don't have to put your device files into /dev, but it's done by convention. Linus put his device files in /dev, and so should you. However, when creating a device file for testing purposes, it's probably OK to place it in your working directory where you compile the kernel module. Just be sure to put it in the right place when you're done writing the device driver. I would like to make a few last points which are implicit from the above discussion, but I'd like to make them explicit just in case. When a device file is accessed, the kernel uses the major number of the file to determine which driver should be used to handle the access. This means that the kernel doesn't really need to use or even know about the minor number. The driver itself is the only thing that cares about the minor number. It uses the minor number to distinguish between different pieces of hardware. By the way, when I say `hardware', I mean something a bit more abstract than a PCI card that you can hold in your hand. Look at these two device files:

*% ls −l /dev/fd0 /dev/fd0u1680*

*brwxrwxrwx 1 root floppy 2, 0 Jul 5 2000 /dev/fd0*

*brw−rw−−−− 1 root floppy 2, 44 Jul 5 2000 /dev/fd0u1680*

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

Chapter 4. Character Device Files

**4.1. Character Device Drivers**

**4.1.1. The file\_operations Structure**

The file\_operations structure is defined in linux/fs.h, and holds pointers to functions defined by the driver that perform various operations on the device. Each field of he structure corresponds to the address of some function defined by the driver to handle a requested operation. For example, every character driver needs to define a function that reads from the device. The file\_operations structure holds the address of the module's function that performs that operation. Here is what the definition looks like for kernel 2.6.5:

*struct file\_operations {*

*struct module \*owner;*

*loff\_t(\*llseek) (struct file \*, loff\_t, int);*

*ssize\_t(\*read) (struct file \*, char \_\_user \*, size\_t, loff\_t \*);*

*ssize\_t(\*aio\_read) (struct kiocb \*, char \_\_user \*, size\_t, loff\_t);*

*ssize\_t(\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*);*

*ssize\_t(\*aio\_write) (struct kiocb \*, const char \_\_user \*, size\_t, loff\_t);*

*int (\*readdir) (struct file \*, void \*, filldir\_t);*

*unsigned int (\*poll) (struct file \*, struct poll\_table\_struct \*);*

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

*int (\*mmap) (struct file \*, struct vm\_area\_struct \*);*

*int (\*open) (struct inode \*, struct file \*);*

*int (\*flush) (struct file \*);*

*int (\*release) (struct inode \*, struct file \*);*

*int (\*fsync) (struct file \*, struct dentry \*, int datasync);*

*int (\*aio\_fsync) (struct kiocb \*, int datasync);*

*int (\*fasync) (int, struct file \*, int);*

*int (\*lock) (struct file \*, int, struct file\_lock \*);*

*ssize\_t(\*readv) (struct file \*, const struct iovec \*, unsigned long, loff\_t \*);*

*ssize\_t(\*writev) (struct file \*, const struct iovec \*, unsigned long, loff\_t \*);*

*ssize\_t(\*sendfile) (struct file \*, loff\_t \*, size\_t, read\_actor\_t, void \_\_user \*);*

*ssize\_t(\*sendpage) (struct file \*, struct page \*, int, size\_t, loff\_t \*, int);*

*unsigned long (\*get\_unmapped\_area) (struct file \*, unsigned long, unsigned long, unsigned long, unsigned long);*

*};*

Some operations are not implemented by a driver. For example, a driver that handles a video card won't need to read from a directory structure. The corresponding entries in the file\_operations structure should be set to NULL. There is a gcc extension that makes assigning to this structure more convenient. You'll see it in modern drivers, and may catch you by surprise. This is what the new way of assigning to the structure looks like:

*struct file\_operations fops = {*

*read: device\_read,*

*write: device\_write,*

*open: device\_open,*

*release: device\_release*

*};*

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

*struct file\_operations fops = {*

*.read = device\_read,*

*.write = device\_write,*

*.open = device\_open,*

*.release = device\_release*

*};*

The meaning is clear, and you should be aware that any member of the structure which you don't explicitly assign will be initialized to NULL by gcc. An instance of struct file\_operations containing pointers to functions that are used to implement read, write, open, ... syscalls is commonly named fops.

**4.1.2. The file structure**

Each device is represented in the kernel by a file structure, which is defined in linux/fs.h. Be aware that a file is a kernel level structure and never appears in a user space program. It's not the same thing as a FILE, which is defined by glibc and would never appear in a kernel space function. Also, its name is a bit misleading; it represents an abstract open `file', not a file on a disk, which is represented by a structure named inode. An instance of struct file is commonly named filp. You'll also see it refered to as struct file file. Resist the temptation. Go ahead and look at the definition of file. Most of the entries you see, like struct dentry aren't used by device drivers, and you can ignore them. This is because drivers don't fill file directly; they only use structures contained in file which are created elsewhere.

**4.1.3. Registering A Device**

As discussed earlier, char devices are accessed through device files, usually located in /dev[7]. The major number tells you which driver handles which device file. The minor number is used only by the driver itself to differentiate which device it's operating on, just in case the driver handles more than one device. Adding a driver to your system means registering it with the kernel. This is synonymous with assigning it a major number during the module's initialization. You do this by using the register\_chrdev function, defined by linux/fs.h.

*int register\_chrdev(unsigned int major, const char \*name, struct file\_operations \*fops);*

where unsigned int major is the major number you want to request, const char \*name is the name of the device as it'll appear in /proc/devices and struct file\_operations \*fops is a pointer to the file\_operations table for your driver. A negative return value means the registration failed. Note that we didn't pass the minor number to register\_chrdev. That's because the kernel doesn't care about the minor number; only our driver uses it. Now the question is, how do you get a major number without hijacking one that's already in use? The easiest way would be to look through Documentation/devices.txt and pick an unused one. That's a bad way of doing things because you'll never be sure if the number you picked will be assigned later. The answer is that you can ask the kernel to assign you a dynamic major number. If you pass a major number of 0 to register\_chrdev, the return value will be the dynamically allocated major number. The downside is that you can't make a device file in advance, since you don't know what the major number will be. There are a couple of ways to do this. First, the driver itself can print the newly assigned number and we can make the device file by hand. Second, the newly registered device will have an entry in /proc/devices, and we can either make the device file by hand or write a shell script to read the file in and make the device file. The third method is we can have our driver make the the device file using the mknod system call after a successful registration and rm during the call to cleanup\_module.

**4.1.4. Unregistering A Device**

We can't allow the kernel module to be rmmod'ed whenever root feels like it. If the device file is opened by a process and then we remove the kernel module, using the file would cause a call to the memory location where the appropriate function (read/write) used to be. If we're lucky, no other code was loaded there, and we'll get an ugly error message. If we're unlucky, another kernel module was loaded into the same location, which means a jump into the middle of another function within the kernel. The results of this would be impossible to predict, but they can't be very positive. Normally, when you don't want to allow something, you return an error code (a negative number) from the function which is supposed to do it. With cleanup\_module that's impossible because it's a void function. However, there's a counter which keeps track of how many processes are using your module. You can see what it's value is by looking at the 3rd field of /proc/modules. If this number isn't zero, rmmod will fail. Note that you don't have to check the counter from within cleanup\_module because the check will be performed for you by the system call sys\_delete\_module, defined in linux/module.c. You shouldn't use this counter directly, but there are functions defined in linux/module.h which let you increase, decrease and display this counter:

*try\_module\_get(THIS\_MODULE): Increment the use count.*

*module\_put(THIS\_MODULE): Decrement the use count.*

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

**4.1.5. chardev.c**

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

*Example 4−1. chardev.c*

*/\* chardev.c: Creates a read−only char device that says how many times*

*\* you've read from the dev file*

*\*/*

*#include <linux/kernel.h>*

*#include <linux/module.h>*

*#include <linux/fs.h>*

*#include <asm/uaccess.h> /\* for put\_user \*/*

*/\* Prototypes − this would normally go in a .h file*

*\*/*

*int init\_module(void);*

*void cleanup\_module(void);*

*static int device\_open(struct inode \*, struct file \*);*

*static int device\_release(struct inode \*, struct file \*);*

*static ssize\_t device\_read(struct file \*, char \*, size\_t, loff\_t \*);*

*static ssize\_t device\_write(struct file \*, const char \*, size\_t, loff\_t \*);*

*#define SUCCESS 0*

*#define DEVICE\_NAME "chardev" /\* Dev name as it appears in /proc/devices \*/*

*#define BUF\_LEN 80 /\* Max length of the message from the device \*/*

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

*\*/*

*static int Major; /\* Major number assigned to our device driver \*/*

*static int Device\_Open = 0; /\* Is device open?*

*/\* Used to prevent multiple access to device \*/*

*static char msg[BUF\_LEN]; /\* The msg the device will give when asked \*/*

*static char \*msg\_Ptr;*

*static struct file\_operations fops = {*

*.read = device\_read,*

*.write = device\_write,*

*.open = device\_open,*

*.release = device\_release*

*};*

*/\* This function is called when the module is loaded*

*\*/*

*int init\_module(void)*

*{*

*Major = register\_chrdev(0, DEVICE\_NAME, &fops);*

*if (Major < 0) {*

*printk(KERN\_ALERT "Registering char device failed with %d\n", Major);*

*return Major;*

*}*

*printk(KERN\_INFO "I was assigned major number %d. To talk to\n", Major);*

*printk(KERN\_INFO "the driver, create a dev file with\n");*

*printk(KERN\_INFO "'mknod /dev/%s c %d 0'.\n", DEVICE\_NAME, Major);*

*printk(KERN\_INFO "Try various minor numbers. Try to cat and echo to\n");*

*printk(KERN\_INFO "the device file.\n");*

*printk(KERN\_INFO "Remove the device file and module when done.\n");*

*return SUCCESS;*

*}*

*/\* This function is called when the module is unloaded*

*\*/*

*void cleanup\_module(void)*

*{*

*/\* Unregister the device*

*\*/*

*int ret = unregister\_chrdev(Major, DEVICE\_NAME);*

*if (ret < 0)*

*printk(KERN\_ALERT "Error in unregister\_chrdev: %d\n", ret);*

*}*

*/\* Methods*

*\*/*

*/\* Called when a process tries to open the device file, like*

*\* "cat /dev/mycharfile"*

*\*/*

*static int device\_open(struct inode \*inode, struct file \*file)*

*{*

*static int counter = 0;*

*if (Device\_Open)*

*return −EBUSY;*

*Device\_Open++;*

*sprintf(msg, "I already told you %d times Hello world!\n", counter++);*

*msg\_Ptr = msg;*

*try\_module\_get(THIS\_MODULE);*

*return SUCCESS;*

*}*

*/\* Called when a process closes the device file.*

*\*/*

*static int device\_release(struct inode \*inode, struct file \*file)*

*{*

*Device\_Open−−; /\* We're now ready for our next caller \*/*

*/\* Decrement the usage count, or else once you opened the file, you'll*

*\* never get get rid of the module.*

*\*/*

*module\_put(THIS\_MODULE);*

*return 0;*

*}*

*/\* Called when a process, which already opened the dev file, attempts to*

*\* read from it.*

*\*/*

*static ssize\_t device\_read(struct file \*filp, /\* see include/linux/fs.h \*/ char \*buffer, /\* buffer to fill with data \*/ size\_t length, /\* length of the buffer \*/ loff\_t \* offset)*

*{*

*/\* Number of bytes actually written to the buffer*

*\*/*

*int bytes\_read = 0;*

*/\* If we're at the end of the message,*

*\* return 0 signifying end of file*

*\*/*

*if (\*msg\_Ptr == 0)*

*return 0;*

*/\* Actually put the data into the buffer*

*\*/*

*while (length && \*msg\_Ptr) {*

*/\* The buffer is in the user data segment, not the kernel*

*\* segment so "\*" assignment won't work. We have to use*

*\* put\_user which copies data from the kernel data segment to*

*\* the user data segment.*

*\*/*

*put\_user(\*(msg\_Ptr++), buffer++);*

*length−−;*

*bytes\_read++;*

*}*

*/\* Most read functions return the number of bytes put into the buffer*

*\*/*

*return bytes\_read;*

*}*

*/\* Called when a process writes to dev file: echo "hi" > /dev/hello*

*\*/*

*static ssize\_t device\_write(struct file \*filp, const char \*buff, size\_t len, loff\_t \* off)*

*{*

*printk(KERN\_ALERT "Sorry, this operation isn't supported.\n");*

*return −EINVAL;*

*}*

**4.1.6. Writing Modules for Multiple Kernel Versions**

The system calls, which are the major interface the kernel shows to the processes, generally stay the same across versions. A new system call may be added, but usually the old ones will behave exactly like they used to. This is necessary for backward compatibility −− a new kernel version is not supposed to break regular processes. In most cases, the device files will also remain the same. On the other hand, the internal interfaces within the kernel can and do change between versions. The Linux kernel versions are divided between the stable versions (n.$<$even number$>$.m) and the development versions (n.$<$odd number$>$.m). The development versions include all the cool new ideas, including those which will be considered a mistake, or reimplemented, in the next version. As a result, you can't trust the interface to remain the same in those versions (which is why I don't bother to support them in this book, it's too much work and it would become dated too quickly). In the stable versions, on the other hand, we can expect the interface to remain the same regardless of the bug fix version (the m number). There are differences between different kernel versions, and if you want to support multiple kernel versions, you'll find yourself having to code conditional compilation directives. The way to do this to compare the macro LINUX\_VERSION\_CODE to the macro KERNEL\_VERSION. In version a.b.c of the kernel, the value of this macro would be $2^{16}a+2^{8}b+c$. While previous versions of this guide showed how you can write backward compatible code with such constructs in great detail, we decided to break with this tradition for the better. People interested in doing such might now use a LKMPG with a version matching to their kernel. We decided to version the LKMPG like the kernel, at least as far as major and minor number are concerned. We use the patchlevel for our own versioning so use LKMPG version 2.4.x for kernels 2.4.x, use LKMPG version 2.6.x for kernels 2.6.x and so on. Also make sure that you always use current, up to date versions of both, kernel and guide.

**Update**: What we've said above was true for kernels up to and including 2.6.10. You might already have noticed that recent kernels look different. In case you haven't they look like 2.6.x.y now. The meaning of the first three items basically stays the same, but a subpatchlevel has been added and will indicate security fixes till the next stable patchlevel is out. So people can choose between a stable tree with security updates and use the latest kernel as developer tree. Search the kernel mailing list archives if you're interested in the full story.