Modules vs Programs

Archived: 2026-04-05 13:09:54 UTC

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/ksyms`.

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[1] 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/ksyms` 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[2]. This memory begins with 0 and extends up to whatever it needs to be. Since the memory space for any two processes don't overlap, every process that can access a memory address, say `0xbffff978`, would be accessing a different location in real physical memory! The processes would be accessing an index named `0xbffff978` which points to some kind of offset into the region of memory set aside for that particular process. For the most part, a process like our Hello, World program can't access the space of another process, although there are ways which we'll talk about later.

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

By the way, I would like to point out that the above discussion is true for any operating system which uses a monolithic kernel[3]. 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.
