Table of Contents, Show Frames, No Frames

Chapter 12
Modules

This chapter describes how the Linux kernel can dynamically load functions, for example filesystems, only when they are needed.

Linux is a monolithic kernel; that is, it is one, single, large program where all the functional components of the kernel have access to all of its internal data structures and routines. The alternative is to have a micro-kernel structure where the functional pieces of the kernel are broken out into separate units with strict communication mechanisms between them. This makes adding new components into the kernel via the configuration process rather time consuming. Say you wanted to use a SCSI driver for an NCR 810 SCSI and you had not built it into the kernel. You would have to configure and then build a new kernel before you could use the NCR 810. There is an alternative, Linux allows you to dynamically load and unload components of the operating system as you need them. Linux modules are lumps of code that can be dynamically linked into the kernel at any point after the system has booted. They can be unlinked from the kernel and removed when they are no longer needed. Mostly Linux kernel modules are device drivers, pseudo-device drivers such as network drivers, or file-systems.

You can either load and unload Linux kernel modules explicitly using the insmod and rmmod commands or the kernel itself can demand that the kernel daemon (`kerneld`) loads and unloads the modules as they are needed.

Dynamically loading code as it is needed is attractive as it keeps the kernel size to a minimum and makes the kernel very flexible. My current Intel kernel uses modules extensively and is only 406Kbytes long. I only occasionally use `VFAT` file systems and so I build my Linux kernel to automatically load the `VFAT` file system module as I mount a `VFAT` partition. When I have unmounted the `VFAT` partition the system detects that I no longer need the `VFAT` file system module and removes it from the system. Modules can also be useful for trying out new kernel code without having to rebuild and reboot the kernel every time you try it out. Nothing, though, is for free and there is a slight performance and memory penalty associated with kernel modules. There is a little more code that a loadable module must provide and this and the extra data structures take a little more memory. There is also a level of indirection introduced that makes accesses of kernel resources slightly less efficient for modules.

Once a Linux module has been loaded it is as much a part of the kernel as any normal kernel code. It has the same rights and responsibilities as any kernel code; in other words, Linux kernel modules can crash the kernel just like all kernel code or device drivers can.

So that modules can use the kernel resources that they need, they must be able to find them. Say a module needs to call `kmalloc()`, the kernel memory allocation routine. At the time that it is built, a module does not know where in memory `kmalloc()` is, so when the module is loaded, the kernel must fix up all of the module's references to `kmalloc()` before the module can work. The kernel keeps a list of all of the kernel's resources in the kernel symbol table so that it can resolve references to those resources from the modules as they are loaded. Linux allows module stacking, this is where one module requires the services of another module. For example, the `VFAT` file system module requires the services of the `FAT` file system module as the `VFAT` file system is more or less a set of extensions to the `FAT` file system. One module requiring services or resources from another module is very similar to the situation where a module requires services and resources from the kernel itself. Only here the required services are in another, previously loaded module. As each module is loaded, the kernel modifies the kernel symbol table, adding to it all of the resources or

symbols exported by the newly loaded module. This means that, when the next module is loaded, it has access to the services of the already loaded modules.

When an attempt is made to unload a module, the kernel needs to know that the module is unused and it needs some way of notifying the module that it is about to be unloaded. That way the module will be able to free up any system resources that it has allocated, for example kernel memory or interrupts, before it is removed from the kernel. When the module is unloaded, the kernel removes any symbols that that module exported into the kernel symbol table.

Apart from the ability of a loaded module to crash the operating system by being badly written, it presents another danger. What happens if you load a module built for an earlier or later kernel than the one that you are now running? This may cause a problem if, say, the module makes a call to a kernel routine and supplies the wrong arguments. The kernel can optionally protect against this by making rigorous version checks on the module as it is loaded.

12.1 Loading a Module

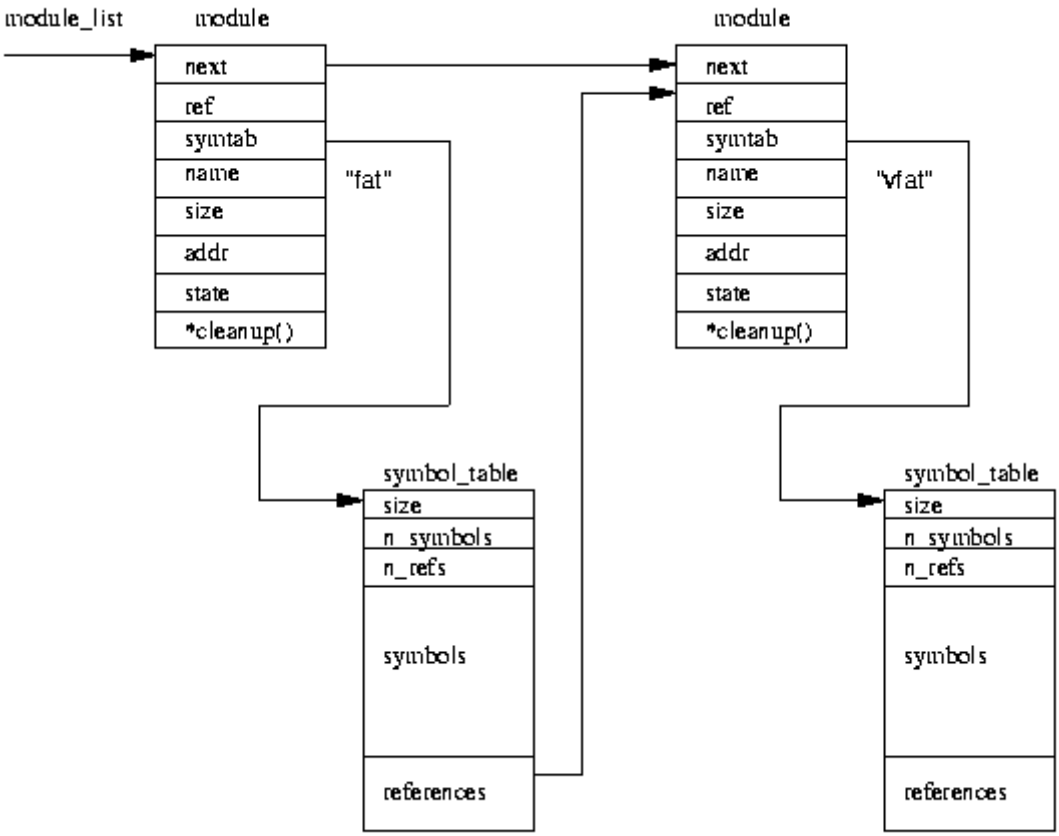

Figure 12.1: The List of Kernel Modules

There are two ways that a kernel module can be loaded. The first way is to use the insmod command to manually insert the it into the kernel.

The second, and much more clever way, is to load the module as it is needed; this is known as demand loading.

When the kernel discovers the need for a module, for example when the user mounts a file system that is not in the kernel, the kernel will request that the kernel daemon (`kerneld`) attempts to load the appropriate module.

The kernel daemon is a normal user process albeit with super user privileges. When it is started up, usually at system boot time, it opens up an Inter-Process Communication (IPC) channel to the kernel. This link is used

by the kernel to send messages to the `kerneld` asking for various tasks to be performed.

`Kerneld`'s major function is to load and unload kernel modules but it is also capable of other tasks such as starting up the PPP link over serial line when it is needed and closing it down when it is not. `Kerneld` does not perform these tasks itself, it runs the neccessary programs such as insmod to do the work. `Kerneld` is just an agent of the kernel, scheduling work on its behalf.

The insmod utility must find the requested kernel module that it is to load. Demand loaded kernel modules are normally kept in `/lib/modules/kernel-version`. The kernel modules are linked object files just like other programs in the system except that they are linked as a relocatable images. That is, images that are not linked to run from a particular address. They can be either `a.out` or `elf` format object files. insmod makes a privileged system call to find the kernel's exported symbols.

These are kept in pairs containing the symbol's name and its value, for example its address. The kernel's exported symbol table is held in the first `module` data structure in the list of modules maintained by the kernel and pointed at by the `module_list` pointer.

Only specifically entered symbols are added into the table, which is built when the kernel is compiled and linked, not *every* symbol in the kernel is exported to its modules. An example symbol is ``` ``request_irq'' ``` which is the kernel routine that must be called when a driver wishes to take control of a particular system interrupt. In my current kernel, this has a value of *0x0010cd30*. You can easily see the exported kernel symbols and their values by looking at `/proc/ksyms` or by using the ksyms utility. The ksyms utility can either show you all of the exported kernel symbols or only those symbols exported by loaded modules. insmod reads the module into its virtual memory and fixes up its unresolved references to kernel routines and resources using the exported symbols from the kernel. This fixing up takes the form of patching the module image in memory. insmod physically writes the address of the symbol into the appropriate place in the module.

When insmod has fixed up the module's references to exported kernel symbols, it asks the kernel for enough space to hold the new kernel, again using a privileged system call. The kernel allocates a new `module` data structure and enough kernel memory to hold the new module and puts it at the end of the kernel modules list. The new module is marked as `UNINITIALIZED`.

Figure 12.1 shows the list of kernel modules after two modules, `VFAT` and `VFAT` have been loaded into the kernel. Not shown in the diagram is the first module on the list, which is a pseudo-module that is only there to hold the kernel's exported symbol table. You can use the command lsmod to list all of the loaded kernel modules and their interdependencies. lsmod simply reformats `/proc/modules` which is built from the list of kernel `module` data structures. The memory that the kernel allocates for it is mapped into the insmod process's address space so that it can access it. insmod copies the module into the allocated space and relocates it so that it will run from the kernel address that it has been allocated. This must happen as the module cannot expect to be loaded at the same address twice let alone into the same address in two different Linux systems. Again, this relocation involves patching the module image with the appropriate addresses.

The new module also exports symbols to the kernel and insmod builds a table of these exported images. Every kernel module must contain module initialization and module cleanup routines and these symbols are deliberately not exported but insmod must know the addresses of them so that it can pass them to the kernel. All being well, insmod is now ready to initialize the module and it makes a privileged system call passing the kernel the addresses of the module's initialization and cleanup routines.

When a new module is added into the kernel, it must update the kernel's set of symbols and modify the modules that are being used by the new module. Modules that have other modules dependent on them must maintain a list of references at the end of their symbol table and pointed at by their `module` data structure. Figure 12.1 shows that the `VFAT` file system module is dependent on the `FAT` file system module. So, the `FAT` module contains a reference to the `VFAT` module; the reference was added when the `VFAT` module was loaded. The kernel calls the modules initialization routine and, if it is successful it carries on installing the module. The module's cleanup routine address is stored in it's `module` data structure and it will be called by the kernel when that module is unloaded. Finally, the module's state is set to `RUNNING`.

12.2 Unloading a Module

Modules can be removed using the rmmod command but demand loaded modules are automatically removed from the system by `kerneld` when they are no longer being used. Every time its idle timer expires, `kerneld` makes a system call requesting that all unused demand loaded modules are removed from the system. The timer's value is set when you start `kerneld`; my `kerneld` checks every 180 seconds. So, for example, if you mount an `iso9660` CD ROM and your `iso9660` filesystem is a loadable module, then shortly after the CD ROM is unmounted, the `iso9660` module will be removed from the kernel.

A module cannot be unloaded so long as other components of the kernel are depending on it. For example, you cannot unload the `VFAT` module if you have one or more `VFAT` file systems mounted. If you look at the output of lsmod, you will see that each module has a count associated with it. For example:

```
Module:        #pages:  Used by:
msdos              5                 1
vfat               4                 1 (autoclean)
fat                6    [vfat msdos]  2 (autoclean)
```

The count is the number of kernel entities that are dependent on this module. In the above example, the `vfat` and `msdos` modules are both dependent on the `fat` module and so it has a count of 2. Both the `vfat` and `msdos` modules have 1 dependent, which is a mounted file system. If I were to load another `VFAT` file system then the `vfat` module's count would become 2. A module's count is held in the first longword of its image.

This field is slightly overloaded as it also holds the `AUTOCLEAN` and `VISITED` flags. Both of these flags are used for demand loaded modules. These modules are marked as `AUTOCLEAN` so that the system can recognize which ones it may automatically unload. The `VISITED` flag marks the module as in use by one or more other system components; it is set whenever another component makes use of the module. Each time the system is asked by `kerneld` to remove unused demand loaded modules it looks through all of the modules in the system for likely candidates. It only looks at modules marked as `AUTOCLEAN` and in the state `RUNNING`. If the candidate has its `VISITED` flag cleared then it will remove the module, otherwise it will clear the `VISITED` flag and go on to look at the next module in the system.

Assuming that a module can be unloaded, its cleanup routine is called to allow it to free up the kernel resources that it has allocated.

The `module` data structure is marked as `DELETED` and it is unlinked from the list of kernel modules. Any other modules that it is dependent on have their reference lists modified so that they no longer have it as a dependent. All of the kernel memory that the module needed is deallocated.

Top of Chapter, Table of Contents, Show Frames, No Frames