FROM I/O PORTS TO PROCESS MANAGEMENT

3rd Edition

Covers Version

Understanding th

Table of Contents

APPENDIX B

Modules

As stated in Chapter 1, *modules* are Linux's recipe for effectively achieving many of the theoretical advantages of microkernels without introducing performance penalties.

To Be (a Module) or Not to Be?

When system programmers want to add new functionality to the Linux kernel, they are faced with a basic decision: should they write the new code so that it will be compiled as a module, or should they statically link the new code to the kernel?

As a general rule, system programmers tend to implement new code as a module. Because modules can be linked on demand (as we see later), the kernel does not have to be bloated with hundreds of seldom-used programs. Nearly every higher-level component of the Linux kernel—filesystems, device drivers, executable formats, network layers, and so on—can be compiled as a module. Linux distributions use modules extensively in order to support in a seamless way a wide range of hardware devices. For instance, the distribution puts tens of sound card driver modules in a proper directory, although only one of these modules will be effectively loaded on a specific machine.

However, some Linux code must necessarily be linked statically, which means that either the corresponding component is included in the kernel or it is not compiled at all. This happens typically when the component requires a modification to some data structure or function statically linked in the kernel.

For example, suppose the component has to introduce new fields into the process descriptor. Linking a module cannot change an already defined data structure such as `task_struct` because, even if the module uses its modified version of the data structure, all statically linked code continues to see the old version. Data corruption easily occurs. A partial solution to the problem consists of "statically" adding the new fields to the process descriptor, thus making them available to the kernel component no matter how it has been linked. However, if the kernel component is never

used, such extra fields replicated in every process descriptor are a waste of memory. If the new kernel component increases the size of the process descriptor a lot, one would get better system performance by adding the required fields in the data structure only if the component is statically linked to the kernel.

As a second example, consider a kernel component that has to replace statically linked code. It's pretty clear that no such component can be compiled as a module, because the kernel cannot change the machine code already in RAM when linking the module. For instance, it is not possible to link a module that changes the way page frames are allocated, because the Buddy system functions are always statically linked to the kernel.*

The kernel has two key tasks to perform in managing modules. The first task is making sure the rest of the kernel can reach the module's global symbols, such as the entry point to its main function. A module must also know the addresses of symbols in the kernel and in other modules. Thus, references are resolved once and for all when a module is linked. The second task consists of keeping track of the use of modules, so that no module is unloaded while another module or another part of the kernel is using it. A simple reference count keeps track of each module's usage.

Module Licenses

The license of the Linux kernel (GPL, version 2) is liberal in what users and industries can do with the source code; however, it strictly forbids the release of source code derived from—or heavily depending on—the Linux code under a non-GPL license. Essentially, the kernel developers want to be sure that their code—and all the code derived from it—will remain freely usable by all users.

Modules, however, pose a threat to this model. Someone might release a module for the Linux kernel in binary form only; for instance, a vendor might distribute the driver for its hardware device in a binary-only module. Nowadays, there are quite a few examples of these practices. Theoretically, characteristics and behavior of the Linux kernel might be significantly changed by binary-only modules, thus effectively turning a Linux-derived kernel in a commercial product.

Thus, binary-only modules are not well received by the Linux kernel developer community. The implementation of Linux modules reflect this fact. Basically, each module developer should specify in the module source code the type of license, by using the `MODULE_LICENSE` macro. If the license is not GPL-compatible (or it is not specified at all), the module will not be able to make use of many core functions and data structures of the kernel. Moreover, using a module with a non-GPL license will

* You might wonder why your favorite kernel component has not been modularized. In most cases, there is no strong technical reason because it is essentially a software license issue. Kernel developers want to make sure that core components will never be replaced by proprietary code released through binary-only "black-box" modules.

Appendix B. Modules. Understanding the Linux Kernel, 3rd Edition, ISBN: 0596005652

"taint" the kernel, which means that any supposed bug in the kernel will not be taken in consideration by the kernel developers.

Module Implementation

Modules are stored in the filesystem as ELF object files and are linked to the kernel by executing the *insmod* program (see the later section, "Linking and Unlinking Modules"). For each module, the kernel allocates a memory area containing the following data:

- A `module` object
- A null-terminated string that represents the name of the module (all modules must have unique names)
- The code that implements the functions of the module

The `module` object describes a module; its fields are shown in Table B-1. A doubly linked circular list collects all `module` objects; the list head is stored in the `modules` variable, while the pointers to the adjacent elements are stored in the `list` field of each `module` object.

Table B-1. The module object

Type	Name	Description
`enum module_state`	`state`	The internal state of the module
`struct list_head`	`list`	Pointers for the list of modules
`char [60]`	`name`	The module name
`struct module_kobject`	`mkobj`	Includes a `kobject` data structure and a pointer to this module object
`struct module_param_attrs *`	`param_attrs`	Pointer to an array of module parameter descriptors
`const struct kernel_symbol *`	`syms`	Pointer to an array of exported symbols
`unsigned int`	`num_syms`	Number of exported symbols
`const unsigned long *`	`crcs`	Pointer to an array of CRC values for the exported symbols
`const struct kernel_symbol *`	`gpl_syms`	Pointer to an array of GPL-exported symbols
`unsigned int`	`num_gpl_syms`	Number of GPL-exported symbols
`const unsigned long *`	`gpl_crcs`	Pointer to an array of CRC values for the GPL-exported symbols
`unsigned int`	`num_exentries`	Number of entries in the module's exception table
`const struct exception_table_entry *`	`extable`	Pointer to the module's exception table
`int (*)(void)`	`init`	The initialization method of the module

Table B-1. The module object (continued)

Type	Name	Description
`void *`	`module_init`	Pointer to the dynamic memory area allocated for module's initialization
`void *`	`module_core`	Pointer to the dynamic memory area allocated for module's core functions and data structures
`unsigned long`	`init_size`	Size of the dynamic memory area required for module's initialization
`unsigned long`	`core_size`	Size of the dynamic memory area required for module's core functions and data structures
`unsigned long`	`init_text_size`	Size of the executable code used for module's initialization; used only when linking the module
`unsigned long`	`core_text_size`	Size of the core executable code of the module; used only when linking the module
`struct mod_arch_specific`	`arch`	Architecture-dependent fields (none in the 80 × 86 architecture)
`int`	`unsafe`	Flag set if the module cannot be safely unloaded
`int`	`license_gplok`	Flag set if the module license is GPL-compatible
`struct module_ref [NR_CPUS]`	`ref`	Per-CPU usage counters
`struct list_head`	`modules_which_use_me`	List of modules that rely on this module
`struct task_struct *`	`waiter`	The process that is trying to unload the module
`void (*)(void)`	`exit`	Exit method of the module
`Elf_Sym *`	`symtab`	Pointer to an array of module's ELF symbols for the */proc/kallsyms* file
`unsigned long`	`num_symtab`	Number of module's ELF symbols shown in */proc/kallsyms*
`char *`	`strtab`	The string table for the module's ELF symbols shown in */proc/kallsyms*
`struct module_sect_attrs *`	`sect_attrs`	Pointer to an array of module's section attribute descriptors (displayed in the *sysfs* filesystem)
`void *`	`percpu`	Pointer to CPU-specific memory areas
`char *`	`args`	Command line arguments used when linking the module

The `state` field encodes the internal state of the module: it can be `MODULE_STATE_LIVE` (the module is active), `MODULE_STATE_COMING` (the module is being initialized), and `MODULE_STATE_GOING` (the module is being removed).

As already mentioned in the section "Dynamic Address Checking: The Fix-up Code" in Chapter 10, each module has its own exception table. The table includes the addresses of the fixup code of the module, if any. The table is copied into RAM when the module is linked, and its starting address is stored in the `extable` field of the `module` object.

Module Usage Counters

Each module has a set of usage counters, one for each CPU, stored in the `ref` field of the corresponding `module` object. The counter is increased when an operation involving the module's functions is started and decreased when the operation terminates. A module can be unlinked only if the sum of all usage counters is 0.

For example, suppose that the MS-DOS filesystem layer is compiled as a module and the module is linked at runtime. Initially, the module usage counters are set to 0. If the user mounts an MS-DOS floppy disk, one of the module usage counters is increased by 1. Conversely, when the user unmounts the floppy disk, one of the counters—even different from the one that was increased—is decreased by 1. The total usage counter of the module is the sum of all CPU counters.

Exporting Symbols

When linking a module, all references to global kernel symbols (variables and functions) in the module's object code must be replaced with suitable addresses. This operation, which is very similar to that performed by the linker while compiling a User Mode program (see the section "Libraries" in Chapter 20), is delegated to the *insmod* external program (described later in the section "Linking and Unlinking Modules").

Some special *kernel symbol tables* are used by the kernel to store the symbols that can be accessed by modules together with their corresponding addresses. They are contained in three sections of the kernel code segment: the `__kstrtab` section includes the names of the symbols, the `__ksymtab` section includes the addresses of the symbols that can be used by all kind of modules, and the `__ksymtab_gpl` section includes the addresses of the symbols that can be used by the modules released under a GPL-compatible license. The `EXPORT_SYMBOL` macro and the `EXPORT_SYMBOL_GPL` macro, when used inside the statically linked kernel code, force the C compiler to add a specified symbol to the `__ksymtab` and `__ksymtab_gpl` sections, respectively.

Only the kernel symbols actually used by some existing module are included in the table. Should a system programmer need, within some module, to access a kernel symbol that is not already exported, he can simply add the corresponding `EXPORT_SYMBOL_GPL` macro into the Linux source code. Of course, he cannot legally export a new symbol for a module whose license is not GPL-compatible.

Linked modules can also export their own symbols so that other modules can access them. The *module symbol tables* are contained in the `__ksymtab`, `__ksymtab_gpl`, and `__kstrtab` sections of the module code segment. To export a subset of symbols from the module, the programmer can use the `EXPORT_SYMBOL` and `EXPORT_SYMBOL_GPL` macros described above. The exported symbols of the module are copied into two memory arrays when the module is linked, and their addresses are stored in the `syms` and `gpl_syms` fields of the `module` object.

Module Dependency

A module (B) can refer to the symbols exported by another module (A); in this case, we say that B is loaded on top of A, or equivalently that A is used by B. To link module B, module A must have already been linked; otherwise, the references to the symbols exported by A cannot be properly linked in B. In short, there is a *dependency* between modules.

The `modules_which_use_me` field of the `module` object of A is the head of a dependency list containing all modules that are used by A; each element of the list is a small `module_use` descriptor containing the pointers to the adjacent elements in the list and a pointer to the corresponding `module` object; in our example, a `module_use` descriptor pointing to the B's `module` object would appear in the `modules_which_use_me` list of A. The `modules_which_use_me` list must be updated dynamically whenever a module is loaded on top of A. The module A cannot be unloaded if its dependency list is not empty.

Beside A and B there could be, of course, another module (C) loaded on top of B, and so on. Stacking modules is an effective way to modularize the kernel source code, thus speeding up its development.

Linking and Unlinking Modules

A user can link a module into the running kernel by executing the *insmod* external program. This program performs the following operations:

1. Reads from the command line the name of the module to be linked.
2. Locates the file containing the module's object code in the system directory tree. The file is usually placed in some subdirectory below */lib/modules*.
3. Reads from disk the file containing the module's object code.
4. Invokes the `init_module( )` system call, passing to it the address of the User Mode buffer containing the module's object code, the length of the object code, and the User Mode memory area containing the parameters of the *insmod* program.
5. Terminates.

The `sys_init_module( )` service routine does all the real work; it performs the following main operations:

1. Checks whether the user is allowed to link the module (the current process must have the `CAP_SYS_MODULE` capability). In every situation where one is adding functionality to a kernel, which has access to all data and processes on the system, security is a paramount concern.

2. Allocates a temporary memory area for the module's object code; then, copies into this memory area the data in the User Mode buffer passed as first parameter of the system call.
3. Checks that the data in the memory area effectively represents a module's ELF object; otherwise, returns an error code.
4. Allocates a memory area for the parameters passed to the *insmod* program, and fills it with the data in the User Mode buffer whose address was passed as third parameter of the system call.
5. Walks the `modules` list to verify that the module is not already linked. The check is done by comparing the names of the modules (`name` field in the `module` objects).
6. Allocates a memory area for the core executable code of the module, and fills it with the contents of the relevant sections of the module.
7. Allocates a memory area for the initialization code of the module, and fills it with the contents of the relevant sections of the module.
8. Determines the address of the `module` object for the new module. An image of this object is included in the `gnu.linkonce.this_module` section of the text segment of the module's ELF file. The `module` object is thus included in the memory area filled in step 6.
9. Stores in the `module_code` and `module_init` fields of the `module` object the addresses of the memory areas allocated in steps 6 and 7.
10. Initializes the `modules_which_use_me` list in the `module` object, and sets to zero all module's reference counters except the counter of the executing CPU, which is set to one.
11. Sets the `license_gplok` flag in the `module` object according to the type of license specified in the module object.
12. Using the kernel symbol tables and the module symbol tables, relocates the module's object code. This means replacing all occurrences of external and global symbols with the corresponding logical address offsets.
13. Initializes the `syms` and `gpl_syms` fields of the `module` object so that they point to the in-memory tables of symbols exported by the module.
14. The exception table of the module (see the section "The Exception Tables" in Chapter 10) is contained in the `__ex_table` section of the module's ELF file, thus it was copied into the memory area allocated in step 6: stores its address in the `extable` field of the `module` object.
15. Parses the arguments of the *insmod* program, and sets the value of the corresponding module variables accordingly.
16. Registers the kobject included in the `mkobj` field of the `module` object so that a new sub-directory for the module appears in the *module* directory of the *sysfs* special filesystem (see the section "Kobjects" in Chapter 13).

17. Frees the temporary memory area allocated in step 2.
18. Adds the `module` object in the `modules` list.
19. Sets the state of the module to `MODULE_STATE_COMING`.
20. If defined, executes the `init` method of the `module` object.
21. Sets the state of the module to `MODULE_STATE_LIVE`.
22. Terminates by returning zero (success).

To unlink a module, a user invokes the *rmmod* external program, which performs the following operations:

1. Reads from the command line the name of the module to be unlinked.
2. Opens the */proc/modules* file, which lists all modules linked into the kernel, and checks that the module to be removed is effectively linked.
3. Invokes the `delete_module()` system call passing to it the name of the module.
4. Terminates.

In turn, the `sys_delete_module()` service routine performs the following main operations:

1. Checks whether the user is allowed to unlink the module (the current process must have the `CAP_SYS_MODULE` capability).
2. Copies the module's name in a kernel buffer.
3. Walks the `modules` list to find the `module` object of the module.
4. Checks the `modules_which_use_me` dependency list of the module; if it is not empty, the function returns an error code.
5. Checks the state of the module; if it is not `MODULE_STATE_LIVE`, returns an error code.
6. If the module has a custom `init` method, the function checks that it has also a custom `exit` method; if no `exit` method is defined, the module should not be unloaded, thus returns an exit code.
7. To avoid race conditions, stops the activities of all CPUs in the system, except the CPU executing the `sys_delete_module()` service routine.
8. Sets the state of the module to `MODULE_STATE_GOING`.
9. If the sum of all reference counters of the module is greater than zero, returns an error code.
10. If defined, executes the `exit` method of the module.
11. Removes the `module` object from the `modules` list, and de-registers the module from the *sysfs* special filesystem.
12. Removes the `module` object from the dependency lists of the modules that it was using.

13. Frees the memory areas that contain the module's executable code, the `module` object, and the various symbol and exception tables.
14. Returns zero (success).

Linking Modules on Demand

A module can be automatically linked when the functionality it provides is requested and automatically removed afterward.

For instance, suppose that the MS-DOS filesystem has not been linked, either statically or dynamically. If a user tries to mount an MS-DOS filesystem, the `mount( )` system call normally fails by returning an error code, because MS-DOS is not included in the `file_systems` list of registered filesystems. However, if support for automatic linking of modules has been specified when configuring the kernel, Linux makes an attempt to link the MS-DOS module, and then scans the list of registered filesystems again. If the module is successfully linked, the `mount( )` system call can continue its execution as if the MS-DOS filesystem were present from the beginning.

The modprobe Program

To automatically link a module, the kernel creates a kernel thread to execute the *modprobe* external program,* which takes care of possible complications due to module dependencies. The dependencies were discussed earlier: a module may require one or more other modules, and these in turn may require still other modules. For instance, the MS-DOS module requires another module named *fat* containing some code common to all filesystems based on a File Allocation Table (FAT). Thus, if it is not already present, the *fat* module must also be automatically linked into the running kernel when the MS-DOS module is requested. Resolving dependencies and finding modules is a type of activity that's best done in User Mode, because it requires locating and accessing module object files in the filesystem.

The *modprobe* external program is similar to *`insmod`*, since it links in a module specified on the command line. However, *`modprobe`* also recursively links in all modules used by the module specified on the command line. For instance, if a user invokes *`modprobe`* to link the MS-DOS module, the program links the *fat* module, if necessary, followed by the MS-DOS module. Actually, *`modprobe`* simply checks for module dependencies; the actual linking of each module is done by forking a new process and executing *`insmod`*.

How does *`modprobe`* know about module dependencies? Another external program named *depmod* is executed at system startup. It looks at all the modules compiled for

* This is one of the few examples in which the kernel relies on an external program.

the running kernel, which are usually stored inside the */lib/modules* directory. Then it writes all module dependencies to a file named *modules.dep*. The *modprobe* program can thus simply compare the information stored in the file with the list of linked modules yielded by the */proc/modules* file.

The request_module( ) Function

In some cases, the kernel may invoke the `request_module( )` function to attempt automatic linking for a module.

Consider again the case of a user trying to mount an MS-DOS filesystem. If the `get_fs_type( )` function discovers that the filesystem is not registered, it invokes the `request_module( )` function in the hope that MS-DOS has been compiled as a module.

If the `request_module( )` function succeeds in linking the requested module, `get_fs_type( )` can continue as if the module were always present. Of course, this does not always happen; in our example, the MS-DOS module might not have been compiled at all. In this case, `get_fs_type( )` returns an error code.

The `request_module( )` function receives the name of the module to be linked as its parameter. It executes `kernel_thread( )` to create a new kernel thread and waits until that kernel thread terminates.

The kernel thread, in turn, receives the name of the module to be linked as its parameter and invokes the `execve( )` system call to execute the *modprobe* external program,* passing the module name to it. In turn, the *modprobe* program actually links the requested module, along with any that it depends on.

* The name and path of the program executed by `exec_modprobe( )` can be customized by writing into the */proc/sys/kernel/modprobe* file.