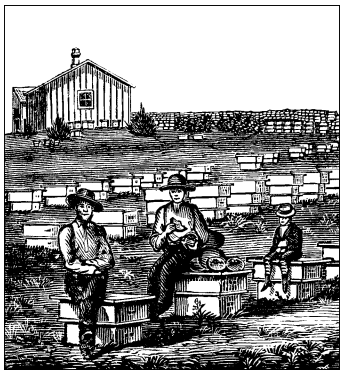
CHAPTER ELEVEN

KMOD AND ADVANCED MODULARIZATION



In this second part of the book, we discuss more advanced topics than we've seen up to now. Once again, we start with modularization.

The introduction to modularization in Chapter 2 was only part of the story; the kernel and the *modutils* package support some advanced features that are more complex than we needed earlier to get a basic driver up and running. The features that we talk about in this chapter include the *kmod* process and version support inside modules (a facility meant to save you from recompiling your modules each time you upgrade your kernel). We also touch on how to run user-space helper programs from within kernel code.

The implementation of demand loading of modules has changed significantly over time. This chapter discusses the 2.4 implementation, as usual. The sample code works, as far as possible, on the 2.0 and 2.2 kernels as well; we cover the differences at the end of the chapter.

Loading Modules on Demand

To make it easier for users to load and unload modules, to avoid wasting kernel memory by keeping drivers in core when they are not in use, and to allow the creation of "generic" kernels that can support a wide variety of hardware, Linux offers support for automatic loading and unloading of modules. To exploit this feature, you need to enable *kmod* support when you configure the kernel before you compile it; most kernels from distributors come with *kmod* enabled. This ability to request additional modules when they are needed is particularly useful for drivers using module stacking.

The idea behind *kmod* is simple, yet effective. Whenever the kernel tries to access certain types of resources and finds them unavailable, it makes a special kernel call to the *kmod* subsystem instead of simply returning an error. If *kmod* succeeds in making the resource available by loading one or more modules, the kernel

continues working; otherwise, it returns the error. Virtually any resource can be requested this way: char and block drivers, filesystems, line disciplines, network protocols, and so on.

One example of a driver that benefits from demand loading is the Advanced Linux Sound Architecture (ALSA) sound driver suite, which should (someday) replace the current sound implementation (Open Sound System, or OSS) in the Linux kernel.* ALSA is split into many pieces. The set of core code that every system needs is loaded first. Additional pieces get loaded depending on both the installed hardware (which sound card is present) and the desired functionality (MIDI sequencer, synthesizer, mixer, OSS compatibility, etc.). Thus, a large and complicated system can be broken down into components, with only the necessary parts being actually present in the running system.

Another common use of automatic module loading is to make a "one size fits all" kernel to package with distributions. Distributors want their kernels to support as much hardware as possible. It is not possible, however, to simply configure in every conceivable driver; the resulting kernel would be too large to load (and very wasteful of system memory), and having that many drivers trying to probe for hardware would be a near-certain way to create conflicts and confusion. With automatic loading, the kernel can adapt itself to the hardware it finds on each individual system.

Requesting Modules in the Kernel

Any kernel-space code can request the loading of a module when needed, by invoking a facility known as *kmod*. *kmod* was initially implemented as a separate, standalone kernel process that handled module loading requests, but it has long since been simplified by not requiring the separate process context. To use *kmod*, you must include `<linux/kmod.h>` in your driver source.

To request the loading of a module, call *request_module*:

```
int request_module(const char *module_name);
```

The `module_name` can either be the name of a specific module file or the name of a more generic capability; we'll look more closely at module names in the next section. The return value from *request_module* will be 0, or one of the usual negative error codes if something goes wrong.

Note that *request_module* is synchronous—it will sleep until the attempt to load the module has completed. This means, of course, that *request_module* cannot be called from interrupt context. Note also that a successful return from *request_module* does not guarantee that the capability you were after is now available. The return value indicates that *request_module* was successful in running *modprobe*,

* The ALSA drivers can be found at *www.alsa-project.org*.

but does not reflect the success status of *modprobe* itself. Any number of problems or configuration errors can lead *request_module* to return a success status when it has not loaded the module you needed.

Thus the proper usage of *request_module* usually requires testing for the existence of a needed capability twice:

```
if ( (ptr = look_for_feature()) == NULL) {
    /* if feature is missing, create request string */
    sprintf(modname, "fmt-for-feature-%i\n", featureid);
    request_module(modname); /* and try lo load it */
}
/* Check for existence of the feature again; error if missing */
if ( (ptr = look_for_feature()) == NULL)
    return -ENODEV;
```

The first check avoids redundant calls to *request_module*. If the feature is not available in the running kernel, a request string is generated and *request_module* is used to look for it. The final check makes sure that the required feature has become available.

The User-Space Side

The actual task of loading a module requires help from user space, for the simple reason that it is far easier to implement the required degree of configurability and flexibility in that context. When the kernel code calls *request_module*, a new "kernel thread" process is created, which runs a helper program in the user context. This program is called *modprobe*; we have seen it briefly earlier in this book.

modprobe can do a great many things. In the simplest case, it just calls *insmod* with the name of a module as passed to *request_module*. Kernel code, however, will often call *request_module* with a more abstract name representing a needed capability, such as `scsi_hostadapter`; *modprobe* will then find and load the correct module. *modprobe* can also handle module dependencies; if a requested module requires yet another module to function, *modprobe* will load both—assuming that *depmod -a* was run after the modules have been installed.*

The *modprobe* utility is configured by the file */etc/modules.conf*.† See the *modules.conf* manpage for the full list of things that can appear in this file. Here is an overview of the most common sorts of entries:

---

* Most distributions run *depmod -a* automatically at boot time, so you don't need to worry about that unless you installed new modules after you rebooted. See the *modprobe* documentation for more details.

† On older systems, this file is often called */etc/conf.modules* instead. That name still works, but its use is deprecated.

`path[misc]=`*`directory`*
: This directive tells *modprobe* that miscellaneous modules can be found in the *misc* subdirectory under the given *`directory`*. Other paths worth setting include `boot`, which points to a directory of modules that should be loaded at boot time, and `toplevel`, which gives a top-level directory under which a tree of module subdirectories may be found. You almost certainly want to include a separate `keep` directive as well.

`keep`
: Normally, a `path` directive will cause *modprobe* to discard all other paths (including the defaults) that it may have known about. By placing a `keep` *before* any `path` directives, you can cause *modprobe* to add new paths to the list instead of replacing it.

`alias` *`alias_name real_name`*
: Causes *modprobe* to load the module *`real_name`* when asked to load *`alias_name`*. The alias name usually identifies a specific capability; it has values such as `scsi_hostadapter`, `eth0`, or `sound`. This is the means by which generic requests ("a driver for the first Ethernet card") get mapped into specific modules. Alias lines are usually created by the system installation process; once it has figured out what hardware a specific system has, it generates the appropriate `alias` entries to get the right drivers loaded.

`options [-k]` *`module opts`*
: Provides a set of options (*`opts`*) for the given *`module`* when it is loaded. If the *-k* flag is provided, the module will not be automatically removed by a *modprobe -r* run.

`pre-install` *`module command`*
`post-install` *`module command`*
`pre-remove` *`module command`*
`post-remove` *`module command`*
: The first two specify a *`command`* to be run either before or after the given *`module`* is installed; the second two run the command before or after module removal. These directives are useful for causing extra user-space processing to happen or for running a required daemon process. The command should be given as a full pathname to avoid possible problems.

  Note that, for the removal commands to be run, the module must be removed with *modprobe*. They will not be run if the module is removed with *rmmod*, or if the system goes down (gracefully or otherwise).

modprobe supports far more directives than we have listed here, but the others are generally only needed in complicated situations.

A typical */etc/modules.conf* looks like this:

```
alias scsi_hostadapter aic7xxx
alias eth0 eepro100
pre-install pcmcia_core /etc/rc.d/init.d/pcmcia start
options short irq=1
alias sound es1370
```

This file tells *modprobe* which drivers to load to make the SCSI system, Ethernet, and sound cards work. It also ensures that if the PCMCIA drivers are loaded, a startup script is invoked to run the card services daemon. Finally, an option is provided to be passed to the *short* driver.

Module Loading and Security

The loading of a module into the kernel has obvious security implications, since the loaded code runs at the highest possible privilege level. For this reason, it is important to be very careful in how you work with the module-loading system.

When editing the *modules.conf* file, one should always keep in mind that anybody who can load kernel modules has complete control over the system. Thus, for example, any directories added to the load path should be very carefully protected, as should the *modules.conf* file itself.

Note that *insmod* will normally refuse to load any modules that are not owned by the root account; this behavior is an attempt at a defense against an attacker who obtains write access to a module directory. You can override this check with an option to *insmod* (or a *modules.conf* line), but doing so reduces the security of your system.

One other thing to keep in mind is that the module name parameter that you pass to *request_module* eventually ends up on the *modprobe* command line. If that module name is provided by a user-space program in any way, it must be very carefully validated before being handed off to *request_module*. Consider, for example, a system call that configures network interfaces. In response to an invocation of *ifconfig*, this system call tells *request_module* to load the driver for the (user-specified) interface. A hostile user can then carefully choose a fictitious interface name that will cause *modprobe* to do something improper. This is a real vulnerability that was discovered late in the 2.4.0-test development cycle; the worst problems have been cleaned up, but the system is still vulnerable to malicious module names.

Module Loading Example

Let's now try to use the demand-loading functions in practice. To this end, we'll use two modules called *master* and *slave*, found in the directory *misc-modules* in the source files provided on the O'Reilly FTP site.

In order to run this test code without installing the modules in the default module search path, you can add something like the following lines to your */etc/modules.conf*:

```
keep
path[misc]=~rubini/driverBook/src/misc-modules
```

The slave module performs no function; it just takes up space until removed. The master module, on the other hand, looks like this:

```
#include <linux/kmod.h>
#include "sysdep.h"


int master_init_module(void)
{
    int r[2]; /* results */

    r[0]=request_module("slave");
    r[1]=request_module("nonexistent");
    printk(KERN_INFO "master: loading results are %i, %i\n", r[0],r[1]);
    return 0; /* success */
}

void master_cleanup_module(void)
{ }
```

At load time, *master* tries to load two modules: the slave module and one that doesn't exist. The *printk* messages reach your system logs and possibly the console. This is what happens in a system configured for *kmod* support when the daemon is active and the commands are issued on the text console:

```
morgana.root# depmod -a
morgana.root# insmod ./master.o
master: loading results are 0, 0
morgana.root# cat /proc/modules
slave                     248   0  (autoclean)
master                    740   0  (unused)
es1370                  34832   1
```

Both the return value from *request_module* and the */proc/modules* file (described in "Initialization and Shutdown" in Chapter 2) show that the slave module has been correctly loaded. Note, however, that the attempt to load `nonexistent` also shows a successful return value. Because *modprobe* was run, *request_module* returns success, regardless of what happened to *modprobe*.

A subsequent removal of *master* will produce results like the following:

```
morgana.root# rmmod master
morgana.root# cat /proc/modules
slave                     248   0  (autoclean)
es1370                  34832   1
```

The *slave* module has been left behind in the kernel, where it will remain until the next module cleanup pass is done (which is often never on modern systems).

Running User-Mode Helper Programs

As we have seen, the *request_module* function runs a program in user mode (i.e., running as a separate process, in an unprivileged processor mode, and in user space) to help it get its job done. In the 2.3 development series, the kernel developers made the "run a user-mode helper" capability available to the rest of the kernel code. Should your driver need to run a user-mode program to support its operations, this mechanism is the way to do it. Since it's part of the *kmod* implementation, we'll look at it here. If you are interested in this capability, a look at *kernel/kmod.c* is recommended; it's not much code and illustrates nicely the use of user-mode helpers.

The interface for running helper programs is fairly simple. As of kernel 2.4.0-test9, there is a function *call_usermodehelper*; it is used primarily by the hot-plug subsystem (i.e., for USB devices and such) to perform module loading and configuration tasks when a new device is attached to the system. Its prototype is:

```
int call_usermodehelper(char *path, char **argv, char **envp);
```

The arguments will be familiar: they are the name of the executable to run, arguments to pass to it (`argv[0]`, by convention, is the name of the program itself), and the values of any environment variables. Both arrays must be terminated by `NULL` values, just like with the *execve* system call. *call_usermodehelper* will sleep until the program has been started, at which point it returns the status of the operation.

Helper programs run in this mode are actually run as children of a kernel thread called *keventd*. An important implication of this design is that there is no way for your code to know when the helper program has finished or what its exit status is. Running helper programs is thus a bit of an act of faith.

It is worth pointing out that truly legitimate uses of user-mode helper programs are rare. In most cases, it is better to set up a script to be run at module installation time that does all needed work as part of loading the module rather than to wire invocations of user-mode programs into kernel code. This sort of policy is best left to the user whenever possible.

Intermodule Communication

Very late in the pre-2.4.0 development series, the kernel developers added a new interface providing limited communication between modules. This intermodule scheme allows modules to register strings pointing to data of interest, which can be retrieved by other modules. We'll look briefly at this interface, using a variation of our *master* and *slave* modules.

We use the same *master* module, but introduce a new slave module called *inter*. All *inter* does is to make a string and a function available under the name `ime_string` (`ime` means "intermodule example") and `ime_function`; it looks, in its entirety, as follows:

```
static char *string = "inter says 'Hello World'";

void ime_function(const char *who)
{
    printk(KERN_INFO "inter: ime_function called by %s\n", who);
}

int ime_init(void)
{
    inter_module_register("ime_string", THIS_MODULE, string);
    inter_module_register("ime_function", THIS_MODULE, ime_function);
    return 0;
}

void ime_cleanup(void)
{
    inter_module_unregister("ime_string");
    inter_module_unregister("ime_function");
}
```

This code uses *inter_module_register*, which has this prototype:

```
void inter_module_register(const char *string, struct module *module,
                           const void *data);
```

`string` is the string other modules will use to find the data; `module` is a pointer to the module owning the data, which will almost always be `THIS_MODULE`; and `data` is a pointer to whatever data is to be shared. Note the use of a `const` pointer for the data; it is assumed that it will be exported in a read-only mode. *inter_module_register* will complain (via *printk*) if the given `string` is already registered.

When the data is no longer to be shared, the module should call *inter_module_unregister* to clean it up:

```
void inter_module_unregister(const char *string);
```

Two functions are exported that can access data shared via *inter_module_register*:

`const void *inter_module_get(const char *string);`
: This function looks up the given `string` and returns the associated data pointer. If the string has not been registered, `NULL` is returned.

```
const void *inter_module_get_request(const char *string,
      const char *module);
```

This function is like *inter_module_get* with the added feature that, if the given `string` is not found, it will call *request_module* with the given `module` name and then will try again.

Both functions also increment the usage count for the module that registered the data. Thus, a pointer obtained with *inter_module_get* or *inter_module_get_request* will remain valid until it is explicitly released. At least, the module that created that pointer will not be unloaded during that time; it is still possible for the module itself to do something that will invalidate the pointer.

When you are done with the pointer, you must release it so that the other module's usage count will be decremented properly. A simple call to

```
void inter_module_put(const char *string);
```

will release the pointer, which should not be used after this call.

In our sample *master* module, we call *inter_module_get_request* to cause the *inter* module to be loaded and to obtain the two pointers. The string is simply printed, and the function pointer is used to make a call from *master* into *inter*. The additional code in *master* looks like this:

```
static const char *ime_string = NULL;
static void master_test_inter();

void master_test_inter()
{
    void (*ime_func)();
    ime_string = inter_module_get_request("ime_string", "inter");
    if (ime_string)
        printk(KERN_INFO "master: got ime_string '%s'\n", ime_string);
    else
        printk(KERN_INFO "master: inter_module_get failed");
    ime_func = inter_module_get("ime_function");
    if (ime_func) {
        (*ime_func)("master");
        inter_module_put("ime_function");
    }
}

void master_cleanup_module(void)
{
    if (ime_string)
        inter_module_put("ime_string");
}
```

Note that one of the calls to *inter_module_put* is deferred until module cleanup time. This will cause the usage count of *inter* to be (at least) 1 until *master* is unloaded.

There are a few other worthwhile details to keep in mind when using the intermodule functions. First, they are available even in kernels that have been configured without support for loadable modules, so there is no need for a bunch of `#ifdef` lines to test for that case. The namespace implemented by the intermodule communication functions is global, so names should be chosen with care or conflicts will result. Finally, intermodule data is stored in a simple linked list; performance will suffer if large numbers of lookups are made or many strings are stored. This facility is intended for light use, not as a general dictionary subsystem.

Version Control in Modules

One of the main problems with modules is their version dependency, which was introduced in Chapter 2. The need to recompile the module against the headers of each kernel version being used can become a real pain when you run several custom modules, and recompiling is not even possible if you run a commercial module distributed in binary form.

Fortunately, the kernel developers found a flexible way to deal with version problems. The idea is that a module is incompatible with a different kernel version only if the software interface offered by the kernel has changed. The software interface, then, can be represented by a function prototype and the exact definition of all the data structures involved in the function call. Finally, a CRC algorithm* can be used to map all the information about the software interface to a single 32-bit number.

The issue of version dependencies is thus handled by mangling the name of each symbol exported by the kernel to include the checksum of all the information related to that symbol. This information is obtained by parsing the header files and extracting the information from them. This facility is optional and can be enabled at compilation time. Modular kernels shipped by Linux distributors usually have versioning support enabled.

For example, the symbol `printk` is exported to modules as something like `printk_R12345678` when version support is enabled, where `12345678` is the hexadecimal representation of the checksum of the software interface used by the function. When a module is loaded into the kernel, *insmod* (or *modprobe*) can accomplish its task only if the checksum added to each symbol in the kernel matches the one added to the same symbol in the module.

There are some limitations to this scheme. A common source of surprises has been loading a module compiled for SMP systems into a uniprocessor kernel, or vice

* CRC means "cyclic redundancy check," a way of generating a short, unique number from an arbitrary amount of data.

versa. Because numerous inline functions (e.g., spinlock operations) and symbols are defined differently for SMP kernels, it is important that modules and the kernel agree on whether they are built for SMP. Version 2.4 and recent 2.2 kernels throw an extra `smp_` string onto each symbol when compiling for SMP to catch this particular case. There are still potential traps, however. Modules and the kernel can differ in which version of the compiler was used to build them, which view of memory they take, which version of the processor they were built for, and more. The version support scheme can catch the most common problems, but it still pays to be careful.

But let's see what happens in both the kernel and the module when version support is enabled:

- In the kernel itself, the symbol is not modified. The linking process happens in the usual way, and the symbol table of the *vmlinux* file looks the same as before.
- The public symbol table is built using the versioned names, and this is what appears in */proc/ksyms*.
- The module must be compiled using the mangled names, which appear in the object files as undefined symbols.
- The loading program (*insmod*) matches the undefined symbols in the module with the public symbols in the kernel, thus using the version information.

Note that the kernel and the module must both agree on whether versioning is in use. If one is built for versioned symbols and the other isn't, *insmod* will refuse to load the module.

Using Version Support in Modules

Driver writers must add some explicit support if their modules are to work with versioning. Version control can be inserted in one of two places: in the makefile or in the source itself. Since the documentation of the *modutils* package describes how to do it in the makefile, we'll show you how to do it in the C source. The *master* module used to demonstrate how *kmod* works is able to support versioned symbols. The capability is automatically enabled if the kernel used to compile the module exploits version support.

The main facility used to mangle symbol names is the header `<linux/modversions.h>`, which includes preprocessor definitions for all the public kernel symbols. This file is generated as part of the kernel compilation (actually, "make depend") process; if your kernel has never been built, or is built without version support, there will be little of interest inside. `<linux/modversions.h>` must be

included before any other header file, so place it first if you put it directly in your driver source. The usual technique, however, is to tell *gcc* to prepend the file with a compilation command like:

```
gcc -DMODVERSIONS -include /usr/src/linux/include/linux/modversions.h...
```

After the header is included, whenever the module uses a kernel symbol, the compiler sees the mangled version.

To enable versioning in the module if it has been enabled in the kernel, we must make sure that `CONFIG_MODVERSIONS` has been defined in `<linux/config.h>`. That header controls what features are enabled (compiled) in the current kernel. Each `CONFIG_` macro defined states that the corresponding option is active.*

The initial part of *master.c*, therefore, consists of the following lines:

```
#include <linux/config.h> /* retrieve the CONFIG_* macros */
#if defined(CONFIG_MODVERSIONS) && !defined(MODVERSIONS)
#  define MODVERSIONS /* force it on */
#endif

#ifdef MODVERSIONS
#  include <linux/modversions.h>
#endif
```

When compiling the file against a versioned kernel, the symbol table in the object file refers to versioned symbols, which match the ones exported by the kernel itself. The following screendump shows the symbol names stored in *master.o*. In the output of *nm*, `T` means "text," `D` means "data," and `U` means "undefined." The "undefined" tag denotes symbols that the object file references but doesn't declare.

```
00000034 T cleanup_module
00000000 t gcc2_compiled.
00000000 T init_module
00000034 T master_cleanup_module
00000000 T master_init_module
         U printk_Rsmp_1b7d4074
         U request_module_Rsmp_27e4dc04
morgana% fgrep 'printk' /proc/ksyms
c011b8b0 printk_Rsmp_1b7d4074
```

Because the checksums added to the symbol names in *master.o* are derived from the entire prototypes of *printk* and *request_module*, the module is compatible with a wide range of kernel versions. If, however, the data structures related to either function get modified, *insmod* will refuse to load the module because of its incompatibility with the kernel.

---

* The `CONFIG_` macros are defined in `<linux/autoconf.h>`. You should, however, include `<linux/config.h>` instead, because the latter is protected from double inclusion, and sources `<linux/autoconf.h>` internally.

Exporting Versioned Symbols

The one thing not covered by the previous discussion is what happens when a module exports symbols to be used by other modules. If we rely on version information to achieve module portability, we'd like to be able to add a CRC code to our own symbols. This subject is slightly trickier than just linking to the kernel, because we need to export the mangled symbol name to other modules; we need a way to build the checksums.

The task of parsing the header files and building the checksums is performed by *genksyms*, a tool released with the *modutils* package. This program receives the output of the C preprocessor on its own standard input and prints a new header file on standard output. The output file defines the checksummed version of each symbol exported by the original source file. The output of *genksyms* is usually saved with a `.ver` suffix; it is a good idea to stay consistent with this practice.

To show you how symbols are exported, we have created two dummy modules called *export.c* and *import.c*. *export* exports a simple function called *export_function*, which is used by the second module, *import.c*. This function receives two integer arguments and returns their sum—we are not interested in the function, but rather in the linking process.

The makefile in the *misc-modules* directory has a rule to build an *export.ver* file from *export.c*, so that the checksummed symbol for *export_function* can be used by the *import* module:

```
ifdef CONFIG_MODVERSIONS
export.o import.o: export.ver
endif

export.ver: export.c
        $(CC) -I$(INCLUDEDIR) $(CFLAGS) -E -D__GENKSYMS__ $^ | \
                $(GENKSYMS) -k 2.4.0 > $@
```

These lines demonstrate how to build *export.ver* and add it to the dependencies of both object files, but only if `MODVERSIONS` is defined. A few lines added to *Makefile* take care of defining `MODVERSIONS` if version support is enabled in the kernel, but they are not worth showing here. The *-k* option must be used to tell *genksyms* which version of the kernel you are working with. Its purpose is to determine the format of the output file; it need not match the kernel you are using exactly.

One thing that is worth showing, however, is the definition of the `GKSMP` symbol. As mentioned above, a prefix (`-p smp_`) is added to every checksum if the kernel is built for SMP systems. The *genksyms* utility does not add this prefix itself; it must be told explicitly to do so. The following makefile code will cause the prefix to be set appropriately:

```
ifdef CONFIG_SMP
        GENKSYMS += -p smp_
endif
```

The source file, then, must declare the right preprocessor symbols for every conceivable preprocessor pass: the input to *genksyms* and the actual compilation, both with version support enabled and with it disabled. Moreover, *export.c* should be able to autodetect version support in the kernel, as *master.c* does. The following lines show you how to do this successfully:

```
#include <linux/config.h> /* retrieve the CONFIG_* macros */
#if defined(CONFIG_MODVERSIONS) && !defined(MODVERSIONS)
#   define MODVERSIONS
#endif

/*
 * Include the versioned definitions for both kernel symbols and our
 * symbol, *unless* we are generating checksums (__GENKSYMS__
 * defined) */
#if defined(MODVERSIONS) && !defined(__GENKSYMS__)
#    include <linux/modversions.h>
#    include "export.ver" /* redefine "export_function" to include CRC */
#endif
```

The code, though hairy, has the advantage of leaving the makefile in a clean state. Passing the correct flags from *make*, on the other hand, involves writing long command lines for the various cases, which we won't do here.

The simple *import* module calls *export_function* by passing the numbers 2 and 2 as arguments; the expected result is therefore 4. The following example shows that *import* actually links to the versioned symbol of *export* and calls the function. The versioned symbol appears in */proc/ksyms*.

```
morgana.root# insmod ./export.o
morgana.root# grep export /proc/ksyms
c883605c export_function_Rsmp_888cb211  [export]
morgana.root# insmod ./import.o
import: my mate tells that 2+2 = 4
morgana.root# cat /proc/modules
import                  312   0  (unused)
export                  620   0  [import]
```

Backward Compatibility

The demand-loading capability was entirely reimplemented in the 2.1 development series. Fortunately, very few modules need to be aware of the change in any way. For completeness, however, we will describe the old implementation here.

In the 2.0 days, demand loading was handled by a separate, user-space daemon process called *kerneld*. This process connected into the kernel via a special interface and received module load (and unload) requests as they were generated by kernel code. There were numerous disadvantages to this scheme, including the fact that no modules could be loaded until the system initialization process had gotten far enough to start *kerneld*.

The *request_module* function, however, remained unchanged, as did all aspects of the modules themselves. It was, however, necessary to include `<linux/kerneld.h>` instead of `<linux/kmod.h>`.

Symbol versioning in the 2.0 kernel did not use the `smp_` prefix on SMP systems. As a result, *insmod* would happily load an SMP module into a uniprocessor kernel, or vice versa. The usual result of such a mismatch was extreme chaos.

The ability to run user-mode helper programs and the intermodule communication mechanism did not exist until Linux 2.4.

Quick Reference

This chapter introduced the following kernel symbols.

`/etc/modules.conf`
: This is the configuration file for *modprobe* and *depmod*. It is used to configure demand loading and is described in the manpages for the two programs.

```
#include <linux/kmod.h>
int request_module(const char *name);
```
: This function performs demand loading of modules.

```
void inter_module_register(const char *string, struct module
      *module, const void *data);
void inter_module_unregister(const char *);
```
: *inter_module_register* makes data available to other modules via the intermodule communication system. When the data is no longer to be shared, *inter_module_unregister* will end that availability.

```
const void *inter_module_get(const char *string);
const void *inter_module_get_request(const char *string,
      const char *module);
 void inter_module_put(const char *string);
```
: The first two functions look up a string in the intermodule communication system; *inter_module_get_request* also attempts to load the given `module` if the string is not found. Both increment the usage count of the module that exported the string; *inter_module_put* should be called to decrement it when the data pointer is no longer needed.

`#include <linux/config.h>`
`CONFIG_MODVERSIONS`
: This macro is defined only if the current kernel has been compiled to support versioned symbols.

`#ifdef MODVERSIONS`
`#include <linux/modversions.h>`
: This header, which exists only if `CONFIG_MODVERSIONS` is valid, contains the versioned names for all the symbols exported by the kernel.

`__GENKSYMS__`
: This macro is defined by *make* when preprocessing files to be read by *genksyms* to build new version codes. It is used to conditionally prevent inclusion of `<linux/modversions.h>` when building new checksums.

```
int call_usermodehelper(char *path, char *argv[], char
    *envp[]);
```
: This function runs a user-mode program in the *keventd* process context.