How To Insert And Remove LKMs

The basic programs for inserting and removing LKMs are **insmod** and **rmmod**. See their man pages for details.

Inserting an LKM is conceptually easy: Just type, as superuser, a command like (`serial.o` contains the device driver for serial ports (UARTs)).

However, I would be misleading you if I said the command just works. It is very common, and rather maddening, for the command to fail either with a message about a module/kernel version mismatch or a pile of unresolved symbols.

If it does work, though, the way to prove to yourself that you know what you're doing is to look at `/proc/modules` as described in Section 5.5.

Note that the examples in this section are from Linux 2.4. In Linux 2.6, the technical aspects of loading LKMs are considerably different, and the most visible manifestation of this is that the LKM file has a suffix of ".ko" instead of ".o". From the user point of view, it looks quite similar, though.

Now lets look at a more difficult insertion. If you try you will probably get a raft of error messages like:

```
msdos.o: unresolved symbol fat_date_unix2dos
msdos.o: unresolved symbol fat_add_cluster1
msdos.o: unresolved symbol fat_put_super
...
```

This is because msdos.o contains external symbol references to the symbols mentioned and there are no such symbols exported by the kernel. To prove this, do a to list every symbol that is exported by the kernel (i.e. available for binding to LKMs). You will see that 'fat_date_unix2dos' is nowhere in the list.

(In Linux 2.6, there is no `/proc/ksyms`. Use `/proc/kallsyms` instead; the format is like the output of **nm**: look for symbols labelled "t").

How do you get it into the list? By loading another LKM, one which defines those symbols and exports them. In this case, it is the LKM in the file `fat.o`. So do and then see that "fat_date_unix2dos" is in `/proc/ksyms`. Now redo the and it works. Look at `/proc/modules` and see that both LKMs are loaded and one depends on the other:

```
msdos                   5632   0 (unused)
fat                    30400   0 [msdos]
```

How did I know `fat.o` was the module I was missing? Just a little ingenuity. A more robust way to address this problem is to use **depmod** and **modprobe** instead of **insmod**, as discussed below.

When your symbols look like "fat_date_unix2dos_R83fb36a1", the problem may be more complex than just getting prerequisite LKMs loaded. See Section 6.

When the error message is "kernel/module version mismatch," see Section 6.

Often, you need to pass parameters to the LKM when you insert it. For example, a device driver wants to know the address and IRQ of the device it is supposed to drive. Or the network driver wants to know how much diagnostic tracing you want it to do. Here is an example of that:

```
insmod ne.o io=0x300 irq=11
```

Here, I am loading the device driver for my NE2000-like Ethernet adapter and telling it to drive the Ethernet adapter at IO address 0x300, which generates interrupts on IRQ 11.

There are no standard parameters for LKMs and very few conventions. Each LKM author decides what parameters **insmod** will take for his LKM. Hence, you will find them documented in the documentation of the LKM. This HOWTO also compiles a lot of LKM parameter information in Section 15. For general information about LKM parameters, see Section 8.

To remove an LKM from the kernel, the command is like

There is a command **lsmod** to list the currently loaded LKMs, but all it does is dump the contents of `/proc/modules`, with column headings, so you may just want to go to the horse's mouth and forget about **lsmod**.

5.1. Could Not Find Kernel Version...

A common error is to try to insert an object file which is not an LKM. For example, you configure your kernel to have the USB core module bound into the base kernel instead of generated as an LKM. In that case, you end up with a file `usbcore.o`, which looks pretty much the same as the `usbcore.o` you would get if you built it as an LKM. But you can't **insmod** that file.

So do you get an error message telling you that you should have configured the kernel to make USB core function an LKM? Of course not. This is Unix, and explanatory error messages are seen as a sign of weakness. The error message is

```
$ insmod usbcore.o
usbcore.o: couldn't find the kernel version this module was compiled for
```

What **insmod** is telling you is that it looked in `usbcore.o` for a piece of information any legitimate LKM would have -- the kernel version with which the LKM was intended to be used -- and it didn't find it. We know now that the reason it didn't find it is that the file isn't an LKM. See Section 10.2 for information on how you can see what **insmod** is seeing and confirm that the file is not in fact an LKM.

If this is a module you created yourself with the intention of it being an LKM, the next question you have is: Why isn't an LKM? The most usual cause of this is that you did not include `linux/module.h` at the top of your source

code and/or did not define the MODULE> macro. MODULE is intended to be set via the compile command (-DMODULE) and determine whether the compilation produces an LKM or an object file for binding into the base kernel. If your module is like most modern modules and can be built *only* as an LKM, then you should just define it in your source code (#define MODULE) before you include include/module.h.

5.2. What Happens When An LKM Loads

So you've successfully loaded an LKM, and verified that via /proc/modules. But how do you know it's working? That's up to the LKM, and varies according to what kind of LKM it is, but here are some of the more common actions of an LKM upon being loaded.

The first thing a device driver LKM does after loading (which is what the module would do at boot time if it were bound into the base kernel) is usually to search the system for a device it knows how to drive. Just how it does this search varies from one driver to the next, and can usually be controlled by module parameters. But in any case, if the driver doesn't find any device it is capable of driving, it causes the load to fail. Otherwise, the driver registers itself as the driver for a particular major number and you can start using the device it found via a device special file that specifies that major number. It may also register itself as the handler for the interrupt level that the device uses. It may also send setup commands to the device, so you may see lights blink or something like that.

You can see that a device driver has registered itself in the file /proc/devices. You can see that the device driver is handling the device's interrupts in /proc/interrupts.

A nice device driver issues kernel messages telling what devices it found and is prepared to drive. (Kernel messages in most systems end up on the console and in the file /var/log/messages. You can also display recent ones with the **dmesg** program). Some drivers, however, are silent. A nice device driver also gives you (in kernel messages) some details of its search when it fails to find a device, but many just fail the load without explanation, and what you get is a list of guesses from **insmod** as to what the problem might have been.

A network device (interface) driver works similarly, except that the LKM registers a device name of its choosing (e.g. eth0) rather than a major number. You can see the currently registered network device names in /proc/net/dev

A filesystem driver, upon loading, registers itself as the driver for a filesystem type of a certain name. For example, the msdos driver registers itself as the driver for the filesystem type named msdos. (LKM authors typically name the LKM the same as the filesystem type it will drive).

5.3. Intelligent Loading Of LKMs - Modprobe

Once you have module loading and unloading figured out using **insmod** and **rmmod**, you can let the system do more of the work for you by using the higher level program **modprobe**. See the **modprobe** man page for details.

The main thing that **modprobe** does is automatically load the prerequisites of an LKM you request. It does this with the help of a file that you create with **depmod** and keep on your system.

Example:

This performs an **insmod** of `msdos.o`, but before that does an **insmod** of `fat.o`, since you have to have `fat.o` loaded before you can load `msdos.o`.

The other major thing **modprobe** does for you is to find the object module containing the LKM given just the name of the LKM. For example, **modprobe msdos** might load `/lib/2.4.2-2/fs/msdos.o`. In fact, **modprobe**'s argument may be a totally symbolic name that you have associated with some actual module. For example, **modprobe eth0** loads the appropriate network device driver to create and drive your `eth0` device, assuming you set that up properly in `modules.conf`. Check out the man pages for **modprobe** and the configuration file `modules.conf` (usually `/etc/modules.conf`) for details on the search rules **modprobe** uses.

modprobe is especially important because it is by default the program that the kernel module loader uses to load an LKM on demand. So if you use automatic module loading, you will need to set up `modules.conf` properly or things will not work. See Section 5.4.

depmod scans your LKM object files (typically all the `.o` files in the appropriate `/lib/modules` subdirectory) and figures out which LKMs prerequire (refer to symbols in) other LKMs. It generates a dependency file (typically named `modules.dep`), which you normally keep in `/lib/modules` for use by **modprobe**.

You can use **modprobe** to remove stacks of LKMs as well.

Via the LKM configuration file (typically `/etc/modules.conf`), you can fine tune the dependencies and do other fancy things to control LKM selections. And you can specify programs to run when you insert and remove LKMs, for example to initialize a device driver.

If you are maintaining one system and memory is not in short supply, it is probably easier to avoid **modprobe** and the various files and directories it needs, and just do raw **insmod**s in a startup script.

5.4. Automatic LKM Loading and Unloading

5.4.1. Automatic Loading

You can cause an LKM to be loaded automatically when the kernel first needs it. You do this with either the kernel module loader, which is part of the Linux kernel, or the older version of it, a `kerneld` daemon.

As an example, let's say you run a program that executes an open system call for a file in an MS-DOS filesystem. But you don't have a filesystem driver for the MS-DOS filesystem either bound into your base kernel or loaded as an LKM. So the kernel does not know how to access the file you're opening on the disk.

The kernel recognizes that it has no filesystem driver for MS-DOS, but that one of the two automatic module loading facilities are available and uses it to cause the LKM to be loaded. The kernel then proceeds with the open.

Automatic kernel module loading is really not worth the complexity in most modern systems. It may make sense in a very small memory system, because you can keep parts of the kernel in memory only when you need them. But the amount of memory these modules uses is so cheap today that you will normally be a lot better off just loading all the modules you might need via startup scripts and leaving them loaded.

Red Hat Linux uses automatic module loading via the kernel module loader.

Both the kernel module loader and `kerneld` use **modprobe**, ergo **insmod**, to insert LKMs. See Section 5.3.

5.4.1.1. Kernel Module Loader

There is some documentation of the kernel module loader in the file `Documentation/kmod.txt` in the Linux 2.4 source tree. This section is more complete and accurate than that file. You can also look at its source code in `kernel/kmod.c`.

The kernel module loader is an optional part of the Linux kernel. You get it if you select the CONFIG_KMOD feature when you configure the kernel at build time.

When a kernel that has the kernel module loader needs an LKM, it creates a user process (owned by the superuser, though) that executes **modprobe** to load the LKM, then exits. By default, it finds **modprobe** as `/sbin/modprobe`, but you can set up any program you like as **modprobe** by writing its file name to `/proc/sys/kernel/modprobe`. For example:

```
# echo "sbin/mymodprobe" >/proc/sys/kernel/modprobe
```

The kernel module loader passes the following arguments to the **modprobe**: Argument Zero is the full file name of **modprobe**. The regular arguments are `-s`, `-k`, and the name of the LKM that the kernel wants. `-s` is the user-hostile form of `--syslog`; `-k` is the cryptic way to say `--autoclean`. I.e. messages from **modprobe** will go to syslog and the loaded LKM will have the "autoclean" flag set.

The most important part of the **modprobe** invocation is, of course, the module name. Note that the "module name" argument to **modprobe** is not necessarily a real module name. It is often a symbolic name representing the role that module plays and you use an `alias` statement in `modules.conf` to tell what LKM gets loaded. For example, if your Ethernet adapter requires the `3c59x` LKM, you would have probably need the line in `/etc/modules.conf`. Here is what the kernel module loader uses for a module name in some of the more popular cases (there are about 20 cases in which the kernel calls on the kernel module loader to load a module):

- When you try access a device and no device driver has registered to serve that device's major number, the kernel requests the module by the name `block-major-N` or `char-major-N` where `N` is the major number in decimal without leading zeroes.
- When you try to access a network interface (maybe by running **ifconfig** against it) and no network device driver has registered to serve an interface by that name, the kernel requests the module named the same as the interface name (e.g. `eth0`). This applies to drivers for non-physical interfaces such as `ppp0` as well.
- When you try to access a socket in a protocol family which no protocol driver has registered to drive, the kernel requests the module named `net-pf-N`, where `N` is the protocol family number (in decimal without leading zeroes).
- When you try to NFS export a directory or otherwise access the NFS server via the NFS system call, the kernel requests the module named `nfsd`.

- The ATA device driver (named `ide`) loads the relevant drivers for classes of ATA devices by the names: `ide-disk`, `ide-cd`, `ide-floppy`, `ide-tape`, and `ide-scsi`.

The kernel module loader runs **modprobe** with the following environment variables (only): HOME=/; TERM=`linux`; PATH=`/sbin:/usr/sbin:/bin:/usr/bin`.

The kernel module loader was new in Linux 2.2 and was designed to take the place of `kerneld`. It does not, however, have all the features of `kerneld`.

In Linux 2.2, the kernel module loader creates the above mentioned process directly. In Linux 2.4, the kernel module loader submits the module loading work to Keventd and it runs as a child process of Keventd.

The kernel module loader is a pretty strange beast. It violates layering as Unix programmers generally understand it and consequently is inflexible, hard to understand, and not robust. Many system designers would bristle just at the fact that it has the PATH hardcoded. You may prefer to use `kerneld` instead, or not bother with automatic loading of LKMs at all.

5.4.1.2. Kerneld

`kerneld` is explained at length in the Kerneld mini-HOWTO, available from the Linux Documentation Project.

`kerneld` is a user process, which runs the kerneld program from the modutils package. `kerneld` sets up an IPC message channel with the kernel. When the kernel needs an LKM, it sends a message on that channel to `kerneld` and `kerneld` runs **modprobe** to load the LKM, then sends a message back to the kernel to say that it is done.

5.4.2. Automatic Unloading - Autoclean

5.4.2.1. The Autoclean Flag

Each loaded LKM has an autoclean flag which can be set or unset. You control this flag with parameters to the `init_module` system call. Assuming you do that via **insmod**, you use the `--autoclean` option.

You can see the state of the autoclean flag in `/proc/modules`. Any LKM that has the flag set has the legend `autoclean` next to it.

5.4.2.2. Removing The Autoclean LKMs

The purpose of the autoclean flag is to let you automatically remove LKMs that haven't been used in a while (typically 1 minute). So by using automatic module loading and unloading, you can keep loaded only parts of the kernel that are presently needed, and save memory.

This is less important than it once was, with memory being much cheaper. If you don't need to save memory, you shouldn't bother with the complexity of module loader processes. Just load everything you might need via an initialization script and keep it loaded.

There is a form of the `delete_module` system call that says, "remove all LKMs that have the autoclean flag set and haven't been used in a while." Kerneld typically calls this once per minute. You can call it explicitly with an **rmmod --all** command.

As the kernel module loader does not do any removing of LKMs, if you use that you might want to have a cron job that does a **rmmod --all** periodically.

5.5. /proc/modules

To see the presently loaded LKMs, do

You see a line like

The left column is the name of the LKM, which is normally the name of the object file from which you loaded it, minus the ".o" suffix. You can, however, choose any name you like with an option on **insmod**.

The "24484" is the size in bytes of the LKM in memory.

The "0" is the use count. It tells how many things presently depend on the LKM being loaded. Typical "things" are open devices or mounted fileystems. It is important because you cannot remove an LKM unless the use count is zero. The LKM itself maintains this count, but the module manager uses it to decide whether to permit an unload.

There is an exception to the above description of the use count. You may see -1 in the use count column. What that means is that this LKM does not use use counts to determine when it is OK to unload. Instead, the LKM has registered a subroutine that the module manager can call that will return an indication of whether or not it is OK to unload the LKM. In this case, the LKM ought to provide you with some custom interface, and some documentation, to determine when the LKM is free to be unloaded.

Do not confuse use count with "dependencies", which are described below.

Here is another example, with more information:

```
lp                      5280   0 (unused)
parport_pc              7552   1
parport                 7600   1 [lp parport_pc]
```

The stuff in square brackets ("[lp parport_pc]") describes dependencies. Here, the modules `lp` and `parport_pc` both refer to addresses within module `parport` (via external symbols that `parport` exports). So `lp` and `parport_pc` are "dependent" on (and are "dependencies of") `parport`.

You cannot unload an LKM that has dependencies. But you can remove those dependencies by unloading the dependent LKMs.

The "(unused)" legend means the LKM has never been used, i.e. it has never been in a state where it could not be unloaded. The kernel tracks this information for one simple reason: to assist in automatic LKM unloading policy. In a system where LKMs are loaded and unloaded automatically (see Section 5.4), you don't want to automatically

load an LKM and then, before the guy who needed it loaded has a chance to use it, unload it because it is not in use.

Here is something you won't normally see:

```
mydriver                8154   0 (deleted)
```

This is an LKM that is in "deleted" state. It's something of a misnomer -- what it means is that the LKM is in the process of being unloaded. You can no longer load LKMs that depend on it, but it's still present in the system. Unloading an LKM is usually close to instantaneous, so if you see this status, you probably have a broken LKM. Its cleanup routine probably got into an infinite loop or stall or crashed (causing a kernel oops). If that's the case, the only way to clear this status is to reboot.

There are similar statuses "initializing" and "uninitialized".

The legend "(autoclean)" refers to the autoclean flag, discussed in Section 5.4.

5.6. Where Are My LKM Files On My System?

The LKM world is flexible enough that the files you need to load could live just about anywhere on your system, but there is a convention that most systems follow: The LKM .o files are in the directory `/lib/modules`, divided into subdirectories. There is one subdirectory for each version of the kernel, since LKMs are specific to a kernel (see Section 6). Each subdirectory contains a complete set of LKMs.

The subdirectory name is the value you get from the **uname --release** command, for example `2.2.19`. Section 6.3 tells how you control that value.

When you build Linux, a standard **make modules** and **make modules_install** should install all the LKMs that are part of Linux in the proper release subdirectory.

If you build a lot of kernels, another organization may be more helpful: keep the LKMs together with the base kernel and other kernel-related files in a subdirectory of /boot. The only drawback of this is that you cannot have /boot reside on a tiny disk partition. In some systems, /boot is on a special tiny "boot partition" and contains only enough files to get the system up to the point that it can mount other filesystems.
