Anatomy of Linux loadable kernel modules

A 2.6 kernel perspective

Skill Level: Intermediate

M. Tim Jones (mtj@mtjones.com)
Consultant Engineer
Emulex Corp.

16 Jul 2008

Linux® loadable kernel modules, introduced in version 1.2 of the kernel, are one of the most important innovations in the Linux kernel. They provide a kernel that is both scalable and dynamic. Discover the ideas behind loadable modules, and learn how these independent objects dynamically become part of the Linux kernel.

The Linux kernel is what's known as a *monolithic kernel,* which means that the majority of the operating system functionality is called the *kernel* and runs in a privileged mode. This differs from a *micro-kernel,* which runs only basic functionality as the kernel (inter-process communication [IPC], scheduling, basic input/output [I/O], memory management) and pushes other functionality outside the privileged space (drivers, network stack, file systems). You'd think that Linux is then a very static kernel, but in fact it's quite the opposite. Linux can be dynamically altered at run time through the use of Linux kernel modules (LKMs).

More in Tim's Anatomy of... series on developerWorks

- Anatomy of Linux flash file systems
- Anatomy of Security-Enhanced Linux (SELinux)
- Anatomy of real-time Linux architectures
- Anatomy of the Linux SCSI subsystem
- Anatomy of the Linux file system
- Anatomy of the Linux networking stack

- Anatomy of the Linux kernel
- Anatomy of the Linux slab allocator
- Anatomy of Linux synchronization methods
- All of Tim's *Anatomy of...* articles
- All of Tim's articles on developerWorks

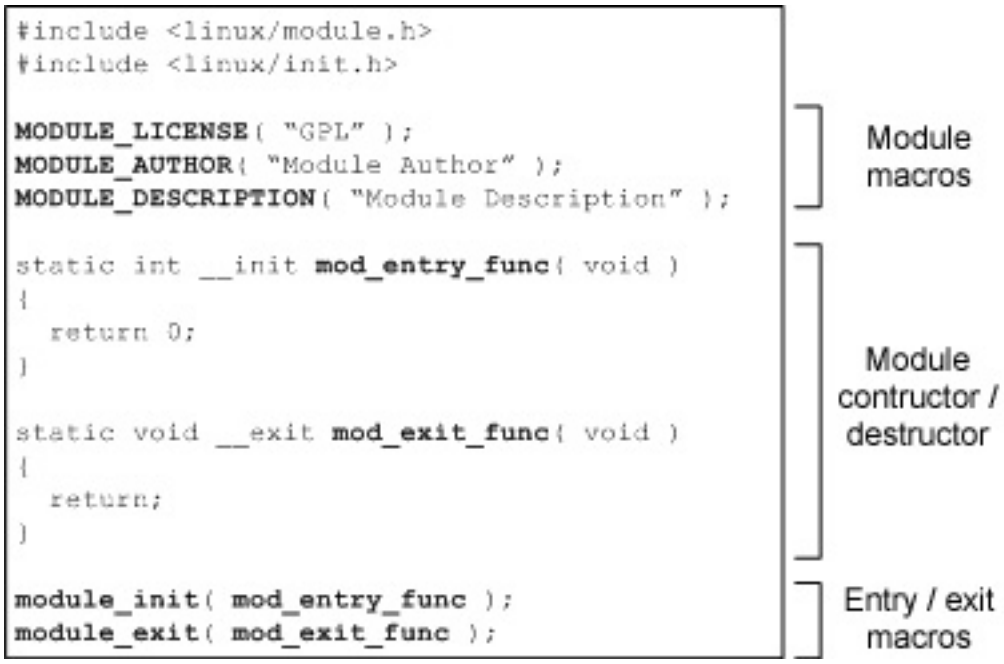
Dynamically alterable means that you can load new functionality into the kernel, unload functionality from the kernel, and even add new LKMs that use other LKMs. The advantage to LKMs is that you can minimize the memory footprint for a kernel, loading only those elements that are needed (which can be an important feature in embedded systems).

Linux is not the only monolithic kernel that can be dynamically altered (and it wasn't the first). You'll find loadable module support in Berkeley Software Distribution (BSD) variants, Sun Solaris, in older kernels such as OpenVMS, and other popular operating systems such as Microsoft® Windows® and Apple Mac OS X.

Anatomy of a kernel module

An LKM has some fundamental differences from elements that compile directly into the kernel and also typical programs. A typical program has a main, where an LKM has a module entry and exit function (in version 2.6, you can name these functions anything you wish). The entry function is called when the module is inserted into the kernel, and the exit function called when it's removed. Because the entry and exit functions are user-defined, a `module_init` and `module_exit` macro exist to define which functions these are. An LKM also includes a required and optional set of module macros. These define the license of the module, the module's author, a description of the module, and more. Figure 1 provides view of a very simple LKM.

Figure 1. Source view of a simple LKM



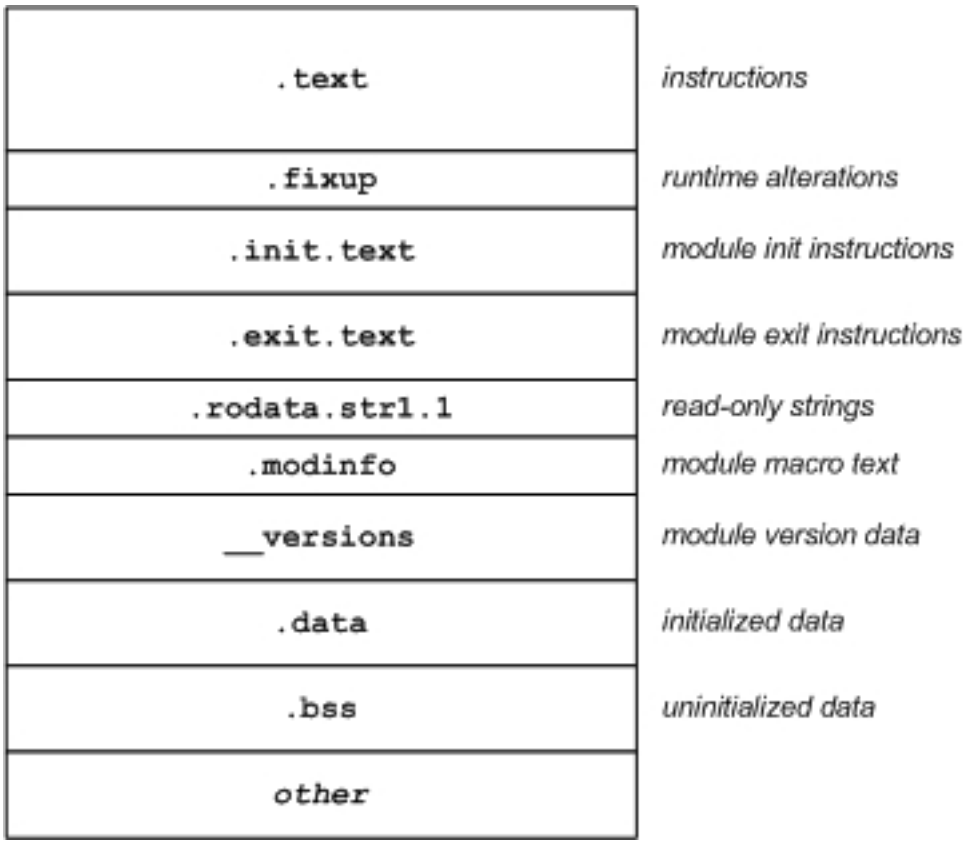
The version 2.6 Linux kernel provides a new (simpler) method for building LKMs. When built, you can use the typical user tools for managing modules (though the internals have changed): the standard `insmod` (installing an LKM), `rmmod` (removing an LKM), `modprobe` (wrapper for `insmod` and `rmmod`), `depmod` (to create module dependencies), and `modinfo` (to find the values for module macros). For more information on building LKMs for the version 2.6 kernel, check out Resources.

Anatomy of a kernel module object

An LKM is nothing more than a special Executable and Linkable Format (ELF) object file. Typically, object files are linked to resolve their symbols and result in an executable. But because an LKM can't resolve its symbols until it's loaded into the kernel, the LKM remains an ELF object. You can use standard object tools on LKMs (which for version 2.6 have the suffix *.ko,* for kernel object). For example, if you used the `objdump` utility on an LKM, you'd find several familiar sections, such as *.text* (instructions), *.data* (initialized data), and *.bss* (Block Started Symbol, or uninitialized data).

You'll also find additional sections in a module to support its dynamic nature. The .init.text section contains the `module_init` code, and the .exit.text contains the `module_exit` code (see Figure 2). The .modinfo section contains the various macro text indicating module license, author, description, and so on.

Figure 2. An example of an LKM with various ELF sections



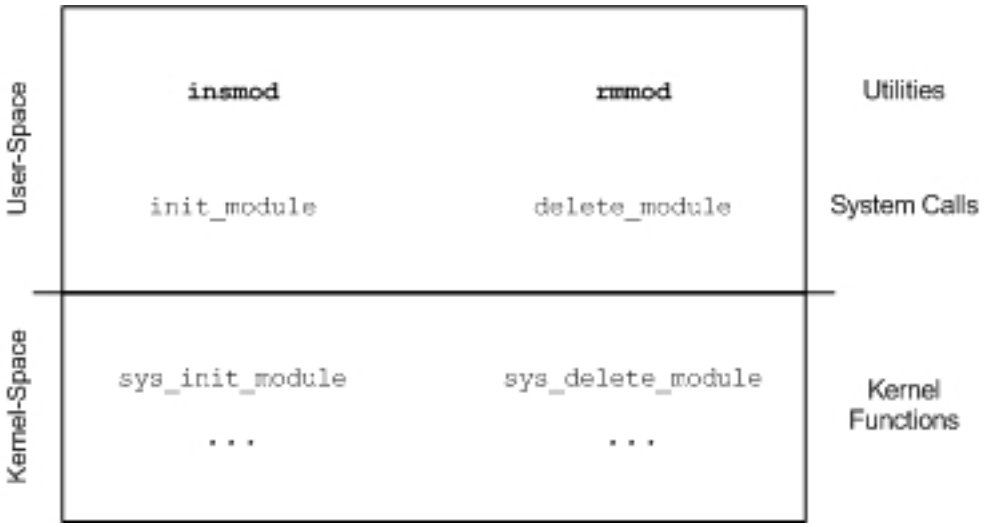
So, with that introduction to the basics of LKMs, let's dig in to see how modules get into the kernel and are managed internally.

Life cycle of an LKM

The process of module loading begins in user space with `insmod` (insert module). The `insmod` command defines the module to load and invokes the `init_module` user-space system call to begin the loading process. The `insmod` command for the version 2.6 kernel has become extremely simple (70 lines of code) based on a change to do more work in the kernel. Rather than `insmod` doing any of the symbol resolution that's necessary (working with `kerneld`), the `insmod` command simply copies the module binary into the kernel through the `init_module` function, where the kernel takes care of the rest.

The `init_module` function works through the system call layer and into the kernel to a kernel function called `sys_init_module` (see Figure 3). This is the main function for module loading, making use of numerous other functions to do the difficult work. Similarly, the `rmmod` command results in a `system call` for `delete_module`, which eventually finds its way into the kernel with a call to `sys_delete_module` to remove the module from the kernel.

Figure 3. Primary commands and functions involved in module loading and unloading

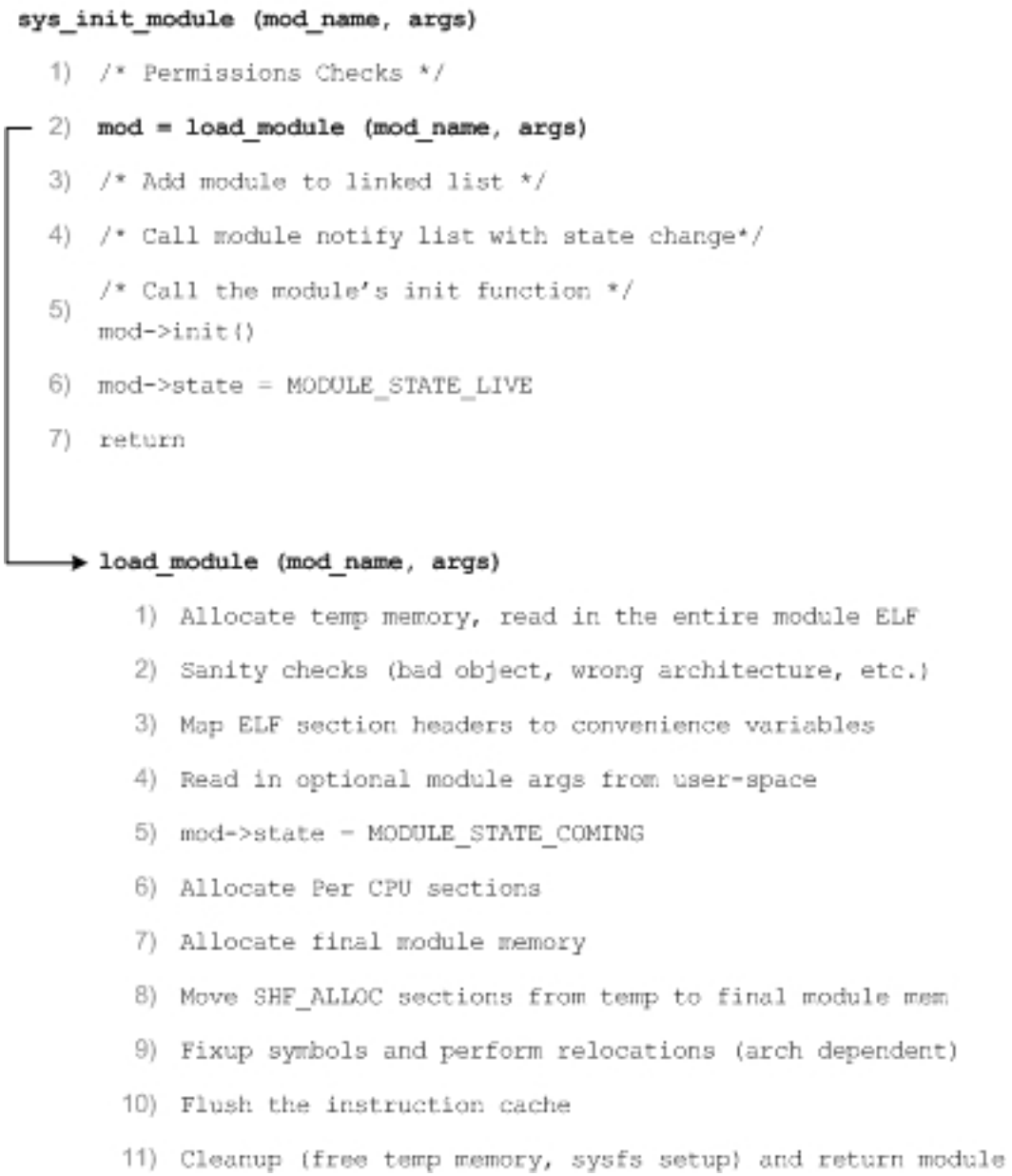


During module load and unload, the module subsystem maintains a simple set of state variables to indicate the operation of a module. If the module is being loaded, then the state is `MODULE_STATE_COMING`. If the module has been loaded and is available, it is `MODULE_STATE_LIVE`. Otherwise, if the module is being unloaded, then the state is `MODULE_STATE_GOING`.

Module loading details

Let's now look at the internal functions for module loading (see Figure 4). When the kernel function `sys_init_module` is called, it begins with a permissions check to see whether the caller can actually perform this operation (through the `capable` function). Then, the `load_module` function is called, which takes care of the mechanical work to bring the module into the kernel and perform the necessary plumbing (I review this shortly). The `load_module` function returns a module reference that refers to the newly loaded module. This module is loaded onto a doubly linked list of all modules in the system, and any threads currently waiting for module state change are notified through the notifier list. Finally, the module's `init()` function is called, and the module's state is updated to indicate that it is loaded and live.

Figure 4. The internal (simplified) module loading process



The internal details of module loading are ELF module parsing and manipulation. The `load_module` function (which resides in ./linux/kernel/module.c) begins by allocating a block of temporary memory to hold the entire ELF module. The ELF module is then read from user space into the temporary memory using `copy_from_user`. As an ELF object, this file has a very specific structure that can be easily parsed and validated.

The next step is to perform a set of sanity checks on the loaded image (is it a valid ELF file? is it defined for the current architecture? and so on). When these sanity checks are passed, the ELF image is parsed and a set of convenience variables are created for each section header to simplify their access later. Because the ELF

objects are based at offset 0 (until relocation), the convenience variables include the relative offset into the temporary memory block. During the process of creating the convenience variables, the ELF section headers are also validated to ensure that a valid module is being loaded.

Any optional module arguments are loaded from user space into another allocated block of kernel memory (step 4), and the module state is updated to indicate that it's being loaded (`MODULE_STATE_COMING`). If per-CPU data is needed (as determined in the section header checks), a per-CPU block is allocated.

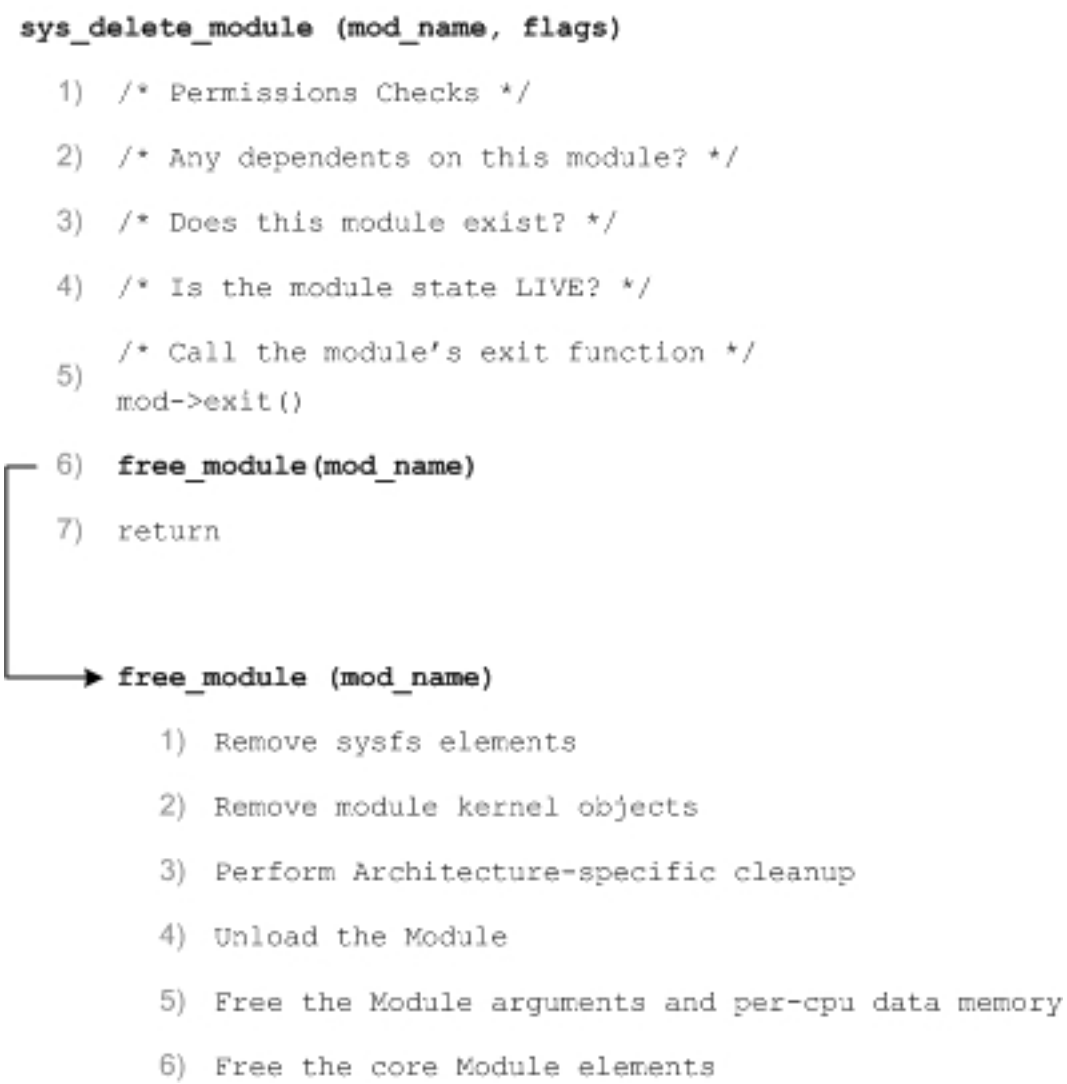
In the prior steps, the module sections are loaded into kernel (temporary) memory, and you also know which are persistent and which can be removed. The next step (7) is to allocate the final location for the module in memory and move the necessary sections (indicated in the ELF headers by `SHF_ALLOC`, or the sections that occupy memory during execution). Another allocation is then performed of the size needed for the required sections of the module. Each section in the temporary ELF block is iterated, and those that need to be around for execution are copied into the new block. This is followed by some additional housekeeping. Symbol resolution also occurs, which can resolve to symbols that are resident in the kernel (compiled into the kernel image) or symbols that are transient (exported from other modules).

The new module is then iterated for each remaining section and relocations performed. This step is architecture dependent and therefore relies on helper functions defined for that architecture (./linux/arch/<arch>/kernel/module.c). Finally, the instruction cache is flushed (because the temporary .text sections were used), a bit more housekeeping is performed (free temporary module memory, setup the sysfs), and the module is finally returned to `load_module`.

Module unloading details

Unloading the module is essentially a mirror of the load process, except that several sanity checks must occur to ensure safe removal of the module. Unloading a module begins in user space with the invocation of the `rmmod` (remove module) command. Inside the `rmmod` command, a system call is made to `delete_module`, which eventually results in a call to `sys_delete_module` inside the kernel (recall from Figure 3). Figure 5 illustrates the basic operation of the module removal process.

Figure 5. The internal (simplified) module unloading process



When the kernel function `sys_delete_module` is invoked (with the name of the module to be removed, passed in as the argument), the first step is to ensure that the caller has permissions. Next, a list is checked to see whether any other modules depend on this module. There exists a list called `modules_which_use_me` that contains an element per dependent module. If this list is empty, no module dependencies exist and the module is a candidate for removal (otherwise, an error is returned). The next test is to see if the module is loaded. Nothing prohibits a user calling `rmmod` on a module that's currently being installed, so this check ensures that the module is live. After a few more housekeeping checks, the penultimate step is to call the module's exit function (provided within the module itself). Finally, the `free_module` function is called.

When `free_module` is called, the module has been found to be safely removable. No dependencies exist now for the module, and the process of cleaning up the kernel can begin for this module. This process begins by removing the module from the various lists that it was placed on during installation (sysfs, module list, and so

on). Next, an architecture-specific cleanup routine is invoked (which can be found in ./linux/arch/<arch>/kernel/module.c). You then iterate the modules that depended on you and remove this module from their lists. Finally, with the cleanup complete—from the kernel's perspective—the various memory that was allocated for the module is freed, including the argument memory, per-CPU memory, and the module ELF memory (`core` and `init`).

Optimizing the kernel for module management

In many applications, the need for dynamic loading of modules is important, but when loaded, it's not necessary for the modules to be unloaded. This allows the kernel to be dynamic at startup (load modules based on the devices that are found) but not dynamic throughout operation. If it's not required to unload a module after it's loaded, you can make several optimizations to reduce the amount of code needed for module management. You can "unset" the kernel configuration option `CONFIG_MODULE_UNLOAD` to remove a considerable amount of kernel functionality related to module unloads.

Going further

This has been a high-level view of the module-management process in the kernel. For the gory details of module management, the best documentation is the source itself. For the main functions involved in module management, see ./linux/kernel/module.c (and the associated header file in ./linux/include/linux/module.h). You can find several architecture-specific functions in ./linux/arch/<arch>/kernel/module.c. Finally, you can see the kernel auto-load function (which automatically loads a module from the kernel based on need) in ./linux/kernel/kmod.c. This feature is enabled through the `CONFIG_KMOD` configuration option.

Resources

Learn

- Follow Rusty Russell's "Bleeding Edge" blog on his current Linux kernel developments. Rusty is the lead developer of the new Linux module architecture.
- The Linux Kernel Module Programming Guide, though a bit dated, provides a great amount of detailed information on LKMs and their development.
- Check out "Access the Linux Kernel using the /proc filesystem" (developerWorks, March 2006) for a detailed look at LKM programming with the /proc file system.
- I Learn more about the details behind system calls in "Kernel command using Linux system calls" (developerWorks, March 2007).
- To learn more about the Linux kernel, Read Tim's "Anatomy of the Linux Kernel (developerWorks, June 2007), the first article in this series, to get a high-level overview of the Linux kernel along with some of its more interesting points.
- Read a great introduction to ELF in "Standards and specs: An unsung hero: the hardworking ELF" (developerWorks, December 2005). The ELF is the standard object format for Linux. ELF is a flexible file format that covers executable images, objects, shared libraries, and even core dumps. You can also find more detailed information in this format reference (PDF document) and detailed book on ELF formats.
- The Captain's Universe provides a great introduction to LKM building with sample makefiles. The process for building LKMs changed with the version 2.6 kernel (for the better).
- There is a small number of module utilities for inserting, removing, and managing modules. Modules are inserted into the kernel with the `insmod` command, and removed with the `rmmod` command. To query the modules currently in the kernel, use the `lsmod` command. Because modules can depend on the presence of other modules, the `depmod` command is available to build a dependency file. To automatically load the dependent modules before the module of interest, you can use the `modprobe` command (a wrapper over `insmod`). Finally, you can read the module information for an LKM using the `modinfo` command.
- The *Linux Journal* article, "Linkers and Loaders" (November 2002) provides a great introduction to the purpose behind linkers and loaders using ELF files (including symbol resolution and relocation).
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.

- See all Linux tips and Linux tutorials on developerWorks.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Order the SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the developerWorks community through blogs, forums, podcasts, and spaces.

About the author

M. Tim Jones
M. Tim Jones is an embedded firmware architect and the author of *Artificial Intelligence: A Systems Approach*, *GNU/Linux Application Programming* (now in its second edition), *AI Application Programming* (in its second edition), and *BSD Sockets Programming from a Multilanguage Perspective.* His engineering background ranges from the development of kernels for geosynchronous spacecraft to embedded systems architecture and networking protocols development. Tim is a Consultant Engineer for Emulex Corp. in Longmont, Colorado.

Trademarks