Practice Tutorial

The Linux kernel uses dynamically loadable modules to simplify and shorten development time, to make configuration easier, and to save kernel memory. This gives a level of flexibility and efficiency not present in many other Unixes. Here's how it's done.

Loadable Kernel Modules

Juan-Mariano de Goyeneche and Elena Apolinario Fernández de Sousa,
Technical University of Madrid

Most Unix kernels are monolithic[1,2]; that is, the kernel is a (normally big) piece of compact code, in which all functions share a common space and are tightly related. When they need to be updated, they must be relinked and reinstalled and the system rebooted before the changes can take effect. This makes modifying them, by adding and testing new drivers, very difficult. The Linux kernel particularly suffered from this problem because of its wide distribution and cooperative development: it was difficult to isolate, test, and integrate the continual stream of changes, enhancements, and additions by people from around the world. Kernel developers soon realized that something had to be done to isolate and track modifications and to avoid frequent kernel builds and reboots.

The community adopted its solution from the microkernel approach for writing operating systems, where many kernel functions are separate user-space components and communicate via microkernel facilities. Although Linux is not a microkernel, it does use loadable modules to simplify and shorten development time, make dynamic configuration easier, and save kernel memory.[3] This gives a level of flexibility and efficiency not present in many other Unixes.

NUTS & BOLTS: Karl Wiegers and Dave Card, editors • kwiegers@acm.org / card@computer.org

Furthermore, even those Unixes that provide a mechanism for dynamically loading modules lack another important Linux feature: the ability to stack modules by following dependencies. This permits code common to a set of similar modules (for example, drivers for similar hardware) to be moved into a single module, so replication is not required. In essence, Linux modules provide a way to dynamically add code to the kernel at runtime, so changes may take place immediately and rebooting is not required. Moreover, when module code is no longer needed, it can be removed, saving kernel memory.

The most recent Linux 2.1.x kernel has modules for most file systems, including several types of CD-ROMs, MS-DOS, Win95, NTFS, NFS, CODA, and ext2 (the de facto standard Linux file system). There are also modules for networking, all high-level SCSI drivers, sound systems, and other purposes. Since the module interface makes it easy to extend the kernel, new modules that support the latest hardware are being developed continuously and added to the default Linux kernel distribution. Modules not yet ready for distribution with the kernel are often distributed via the Web.

This article explains how Linux modules are implemented. To learn how to write your own kernel modules, see our reference list.[4-8]

How Modules Work

Suppose from time to time a user needs to access data from an ISO9660 CD-ROM. Many users use CD-ROMs only occasionally. The ISO9660 file system occupies about 20 Kbytes of kernel code. So unless the module would be frequently loaded and unloaded, it would be a good idea to compile the kernel with the ISO9660 file system as a module.

Next time the system administrator wants to mount a CD-ROM, she could insert that module with the command `# insmod isofs`. This would link the module to the running kernel (similar to the way the standard linker, `ld`, links object files to produce executables). Modules run in privileged mode (as part of the kernel), and must therefore be inserted by the system administrator (root). It would be a huge security hole if ordinary users could insert code into a running kernel. Any attempt to mount a CD-ROM before its file system module is loaded would result in an error, since the kernel would not recognize the underlying file system. Once a module is unneeded, it can be removed from the kernel with the command `# rmmod isofs`. This also frees its memory and releases resources.

Modules can also be linked to other modules, introducing dependencies. Module stacking occurs when one module requires another's services. Frame grabbers, sound cards, and other device drivers are often stacked. Some network protocols also stack modules; for example, the point-to-point protocol module depends on the `slhc` module, which implements Van Jacobson's routines to compress and uncompress TCP packets for transmission over low-speed serial lines. Since the `slhc` code is a module, it is available to other protocols which would otherwise need to duplicate that code, wasting memory and making errors more likely.

To illustrate module dependencies, suppose the `bttv` frame grabber driver is loaded directly.

```
# insmod bttv
../bttv.o: unresolved symbol
   i2c_unregister_bus
../bttv.o: unresolved symbol
   video_register_device
../bttv.o: unresolved symbol
   video_unregister_device
../bttv.o: unresolved symbol
   i2c_register_bus
../bttv.o: unresolved symbol
   i2c_control_device
```

The `bttv` driver uses code from the `i2c` and `videodev` modules. Thus, these modules need to be loaded before all the symbols referenced by `bttv` can be resolved. Finally, `bttv` itself can be installed.

However, it is not necessary to know all dependencies, or to load them one by one. The utility `modprobe` will automatically determine the dependencies and load all the required modules.

The `lsmod` utility can be used to determine the modules currently loaded in the kernel. For the example given earlier, here is what might result:

```
# lsmod
Module          Pages    Used by
bttv              7          0
i2c               1   [bttv]  0
videodev          1   [bttv]  2
```

Here are clear dependencies: `i2c` and `videodev` are being used by `bttv`. Alternatively, the special Linux `/proc` file system provides a window into the current status of the OS and hardware: the `cat /proc/modules` command provides the same information as `lsmod`.

Another possible situation is when a single

module is used by several others. In the following example, `sound` is referenced by `opl3`, `sb`, and `uart401`. The module `sb` is also using `uart401`.

```
# lsmod
Module    Pages  Used by
opl3      3      0
sb        6      0
uart401   2      [sb]  0
sound     16     [opl3 sb uart401] 0
```

Despite the flexibility that modules provide, it is still tedious to require the privileged root account to load and unload them every time a file system or driver needs to be accessed. There is an easier way: the 2.0.x kernels may be compiled with the `Kernel daemon support (e.g., autoload of modules)` option, and the 2.1.x kernels may be built with the `Kernel module loader` option selected. With autoloading turned on, Linux will try to load the appropriate module when a capability is not found within the currently loaded code. Unlike many other Unixes, this takes place not only when opening special files from the `/dev` directory but also whenever Linux searches for an internal feature that is not found inside the built kernel (protocol families, file systems, and so on).

For 2.0.x kernels, the user space daemon `kerneld` receives queries from the kernel and inserts the needed modules by using `modprobe`. However, in recent 2.1.x series kernels, `kerneld` is not used; the task is performed internally by the `kmod` kernel thread, which also runs `modprobe` to insert needed capabilities. (For more on kernel version problems, see the boxed text on this page.) Mounting a CD-ROM is straightforward:

```
# mount -t iso9660 /dev/cdrom /cdrom
# lsmod
Module Pages  Used by
isofs  5      1 (autoclean)
```

This causes the `isofs` module to load automatically when the kernel detects it needs the ISO9660 file system, and no error messages are generated this time.

Also note the `autoclean` label. This means the module was not directly loaded by `insmod` or `modprobe`, but as a consequence of a kernel request. So, when the use count drops to zero (when no one is using the module; that is, when the CD-ROM is unmounted in our example), the kernel will give it a grace period, after which the module will be unloaded if still not in use, without user intervention. (This behavior is true for 2.0.x kernels; in the 2.1.x series it was dropped in favor of less unnecessary code in kernel space. Unused modules can also be unloaded every few minutes using `cron`.) You don't even need to mount and unmount the CD-ROM as root in order to have the module loaded and unloaded. If you configure `/etc/fstab` so that any user can mount or unmount the CD-ROM drive, modules will still load and unload automatically. Together, these features make the use of modules transparent to the user.

The Problem with Kernel Versions

Despite their many benefits, kernel modules also introduced some problems. Since it is possible to compile the kernel and the modules separately, it is also possible to compile them from different source trees. Suppose a module calls a kernel function whose prototype has changed with newer versions of the kernel. Combining the two mismatched codes could cause a system crash, or perhaps something worse. `insmod` cannot detect such errors; it knows only about symbol names and associated addresses. Function parameters are not described by that information.

One way to avoid this kind of problem is to store in the module the version of the kernel headers used to compile it. That version information can then be checked against the running kernel before the module is inserted. If the versions don't match, `insmod` gives an error and exits.

However, this approach is not flexible enough. Another ingenious solution is to perform a 32-bit CRC (Cyclic Redundancy Code) on each variable, function prototype, and data structure. Symbol names are then mangled with the hexadecimal representation of the CRC—giving, for example, `jiffies_R2f7c7437` or `printk_Rad1148ba`.

When inserting modules, `insmod` compares the symbols' CRCs. If they match, the variable definition/interface has not changed, and the module may be safely inserted in the usual way. Both the kernel and the module must be compiled with version information for this solution to be effective.

Internal Design and Implementation Details

Since module autoloading is extensively documented elsewhere,[9,10] we will focus on module stacking. For the purposes of this article, we will discuss kernel version 2.1.125, the most current when we wrote this article. By the time you read this, version 2.2 may have been released. Nevertheless, it will probably not differ significantly from 2.1.125. During the editing phase of this article, we confirmed that version 2.2.0-pre4 has no significant differences, apart from a special treatment for the `usecount` field in order to make its changes atomic.

First, five new system calls were added to the

kernel: `create_module()`, `init_module()`, `delete_module()`, `query_module()`, and `get_kernel_syms()`. From kernel v2.1.18 on, the latter syscall is not supported; `query_module()` should be used instead. The system calls will return the `-ENOSYS` error if support for kernel modules was not selected when the kernel was compiled.

As explained earlier, `insmod` links modules to the kernel. That is, it searches for references to functions and variables (from now on, we will call these *symbols*) not resolved by the linker when it created the object file, and tries to resolve them with the kernel memory addresses associated with those symbols. To do so, the kernel maintains a symbol table—a list of symbols and their addresses. You can display the current symbol table by again looking at the `/proc` file system with `cat /proc/ksyms`. To determine what external references remain unresolved for a given module, use `nm your_module.o`. Symbols preceded by a "U" are unresolved.

The first task `insmod` performs after determining the module it wants to insert is to retrieve the symbol table via the `query_module()` system call. `insmod` first fetches symbols from modules already loaded, then it gets the kernel's symbols.

`query_module()` is passed a buffer, where it writes module names or symbol tables, depending on the query. The value result argument, `ret`, holds one of two values: if the buffer passed into the function is large enough to hold the result, `ret` returns the number of symbols or names stored in the buffer; if the buffer was too small, `query_module()` returns an error, and `ret` provides the minimum size needed, so the buffer might be reallocated.

Therefore, the get_kernel_symbols algorithm might be summarized as follows:

```
get the total number of modules currently loaded,
    and their names;
for each module:
    query the kernel (passing its name) to get some
        info about it (address in memory, size, flags,
        use_count);
    query the kernel to get its sym-tab;
    get global kernel symbol table.
```

After all the symbols and associated addresses have been retrieved, the module must be patched. As can be shown by doing a `file your_module.o`, a module is a *relocatable* ELF (Executable and Linkable Format, the standard executable file format on many Unixes).[11]

ELFs are divided into sections; some of those sections are loaded directly into memory and some are not. `insmod` modifies and adds required information before the ELF module can be loaded.

Note that the patch is done in memory: unresolved symbols, such as `printk`, are matched with their current kernel memory position. The task is easily accomplished by means of two library functions: `obj_find_symbol()` and `obj_add_symbol()`. The idea is as follows:

```
for each module loaded "i":
    for each symbol exported by the module:
        call obj_find_symbol(), passing the symbol's
            name, to see whether the symbol is refer-
            enced in the module;
        if it is:
            call obj_add_symbol(), with arguments
                the symbol's name and its current
                memory address. This places the right
                address for the symbol;
            mark the module "i" as used by the to-be-
                inserted one. This information will be
                necessary when we build the depen-
                dencies table later on;
for each global kernel symbol:
    call obj_find_symbol() and obj_add_symbol()
        with identical semantics.
```

At this point, the module's memory image with all kernel-space references are correct, and are pointing to the right addresses. Unresolved references at this time indicate an error, which would mean that the module cannot be loaded.

It is often useful to pass command line arguments (such as IRQ numbers or I/O addresses) when modules are loaded. So at this time, module arguments are passed: any int or char* global variables can be set with `insmod` at load time.

The loading mechanism

More interesting is the next step—the loading mechanism. It is time to prepare the module's symbol table to permit access by modules that may be inserted in the future.

At this point, we need to know the specific data structures. As shown in Figure 1, each module is defined by a module structure. We introduce some basic fields, such as the module's name, its size, the pointer to the next module in the linked list of

modules, or its number of symbols and dependencies. Two additional structures are also worth noting: `module_symbol{}`, used to place the module's exported symbol table, and `module_ref{}`,which plays a primary role in keeping dependency information.

To construct the symbol table information, `insmod` scans the ELF memory image again; not all ELF sections are loaded, so `insmod` must first determine which are, then get the correspondent symbols from them and add those to the symbol table that will be pointed by the `syms` `struct module` data member. (Note that at this time, `syms` is not pointing to the table.) Now `insmod` creates a `__ksymtab` ELF section and places all the exported symbols there.

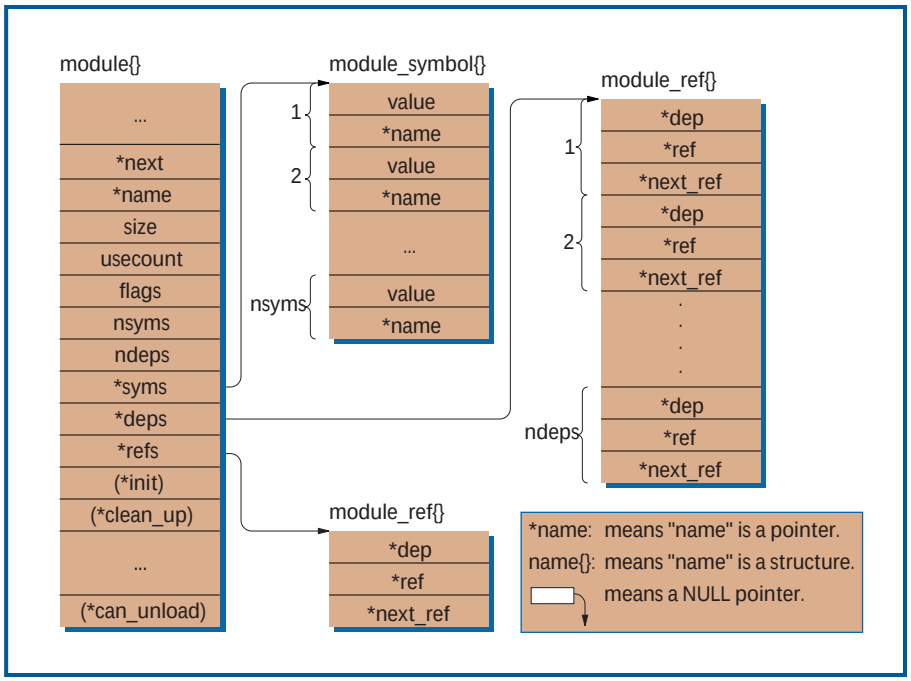

Figure 1. Basic kernel data structures for the modules implementation.

Before we can insert the module, we must build the module dependencies (that is, which already loaded modules the new module will use) so the kernel does not unload any of them while another module is using their services. The `deps` and `refs` pointers into the `module` structure provide this functionality: `deps` traces the modules it depends on (those it needs to run), while `refs` tells which modules need this one (which ones reference it).

Dependency tracking is quite simple. When `insmod` traversed the modules linked list and patched unresolved references, it marked existing modules that were going to be used by the one being inserted. This time we only need to allocate another section, `.kmodtab`, and travel through the list again:

for each module:
    if it is marked "used":
        fill a module_ref{} into the .kmodtab section.

This `module_ref` structure is filled according to the following criteria: `dep` is made to point to the module used (remember that earlier, we got all module memory addresses via the get_kernel_symbols algorithm), while `next_ref` is set to null. `ref` can't be set by `insmod`: it will be manipulated by the kernel when new modules reference this one.

The `create_module()` syscall is invoked next, passing the module's name and its final size. With this information the kernel makes sure no other module with the same name already exists, and allocates enough space to hold the module.

At this point, most of the structures depicted in Figure 1, as well as `module_ref` and `module_symbol`, are complete, though not yet linked to `module{}`, which still has some empty fields. To fill those fields the `.this`, `__ksymtab`, and `.kmodtab` sections and addresses are found and linked by `insmod` to the `module` structure itself, and to the `syms` and `deps` pointers respectively. Some symbols, such as `init_module` and `cleanup_module`, are also searched and their memory addresses assigned to the `init` and `cleanup` fields. If `insmod` was called with the `-k/–autoclean`, the module's `MOD_AUTOCLEAN` flag is set, so it can be "automagically" deleted if it has not been used for a while. `insmod`'s work is nearly done. It calls the `init_module()` system call and lets the kernel do the rest.

This call is perhaps the most intricate, so it will be explained with several pictures. It receives the module's name and a pointer to the module's image, with the module structure on top of it.

After a sanity check (a comment in the source code reads "OK, that's about all the sanity we can stomach; copy the rest"), the kernel copies the image from user space to kernel space.

Since `create_module()` already placed it into the linked list, only dependencies and references must be properly linked now. So the kernel scans the dependency table (pointed to by `deps`). For each of its entries, it traverses the complete list of modules to make sure the needed modules are still there; every time the module referenced in the `deps` table

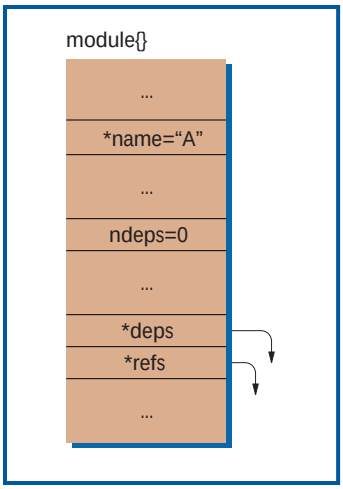

Figure 2. State after module A is inserted.

is found in the list, it updates the dependencies.

To illustrate this, suppose we have already inserted a simple module, A, which does not depend on any module but itself, and is not yet referenced by any other module. Its situation would resemble that in Figure 2.

Now, the system administrator inserts a new module, B. B depends only on A, so this time `deps` is not null (see Figure 3). In its `module_ref` associated structure, `dep` points to the module that B depends on (that is, A). B's `deps->ref` points to the module that B is using. After all, B needs B to run, so it points to itself.

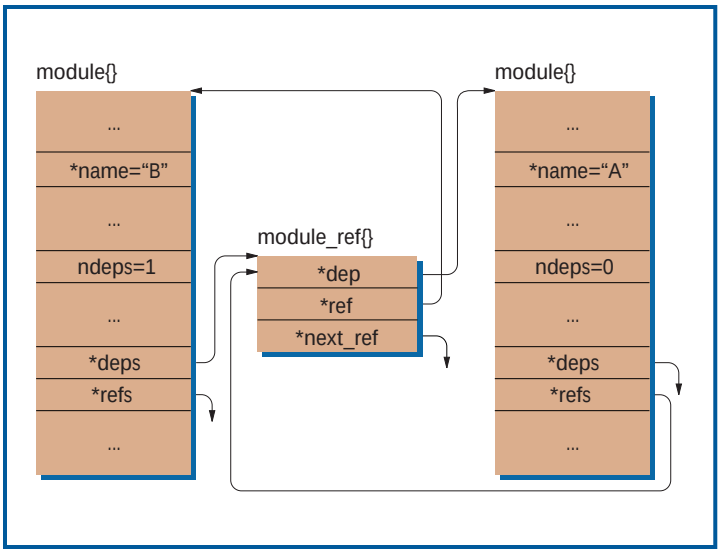

Figure 3. State after module B is inserted. B depends on A.

Finally, let's introduce a third module, which calls some code both from A and B. `insmod` notices it while building C's references, and thus places two entries into the dependencies table, making the first `dep` point to B and the second to A. As usual, `ref` is made to point to C in both entries of its `dep` table. Now, C's `deps->next_ref` pointer is put to the value that B's `refs` had in Figure 3, that is, to null. B's `refs` is replaced with C's `deps` contents, thus pointing to C's `deps->dep`.

Module C also depends on A, so the kernel keeps scanning the modules linked list until it finds A (the scan is done to assure that module A is still loaded).

The same operations are repeated here: C's `(deps+1)->next_ref` is assigned A's `refs` content, thus pointing to B's dependency table. A's `refs` is immediately changed so that it points to C's `(deps+1)->dep`.

Suppose we want to know what modules C is using. Its `deps` field leads us to its dependency table, whose first `dep` points to B and the second one to A. If `p` were a pointer to C, we would retrieve B's and A's names with `p->deps->dep->name` and `p->(deps+1)->dep->name` respectively.

If, on the other hand, we were interested in getting the names of all modules that need A to run, we would get a pointer to A (say its name is q). C's name would be reached via `q->refs->ref->name`. As `q->refs->next_ref` is not null, more modules use it: the first we'd find would be `q->refs->next_ref->ref->name`. As this time `q->refs->next_ref->next_ref` is null, no more modules depend on A, and we are finished. When modules are deleted, this procedure is inverted, replacing all occurrences of `refs` with `refs->next_refs`.

Linux kernel modules provide a powerful mechanism for both kernel developers and end users. If you have ever written a device driver for an operating system without modules, and had to relink and reboot each time you changed something, you'll certainly appreciate them. Modules also help to keep the kernel's memory image small, by only loading those parts that are needed.

Although the implementation might seem cumbersome at times, especially when you look at Figure 4, it is very efficient. The kernel developers have managed to make all the required variable assignments in just four instructions.

In the future, modules might be extended to cope with even more parts of the kernel, such as memory management. This will take place sooner or later thanks to the free-software spirit and open attitude. ❖

Acknowledgment

We thank Javier Macías Guarasa for his continuous encouragement.

References

1. M.J. Bach, *The Design of the UNIX Operating System*, Prentice Hall, Englewood Cliffs, N.J., 1986.
2. M.K. McKusick et al., "The Design and Implementation of the 4.4 BSD UNIX Operating System," Addison Wesley Longman, Reading, Pa., 1996.
3. Linux kernel source code: ftp://ftp.kernel.org, files in linux/kernel/module.c and linux/include/ linux/module.h.

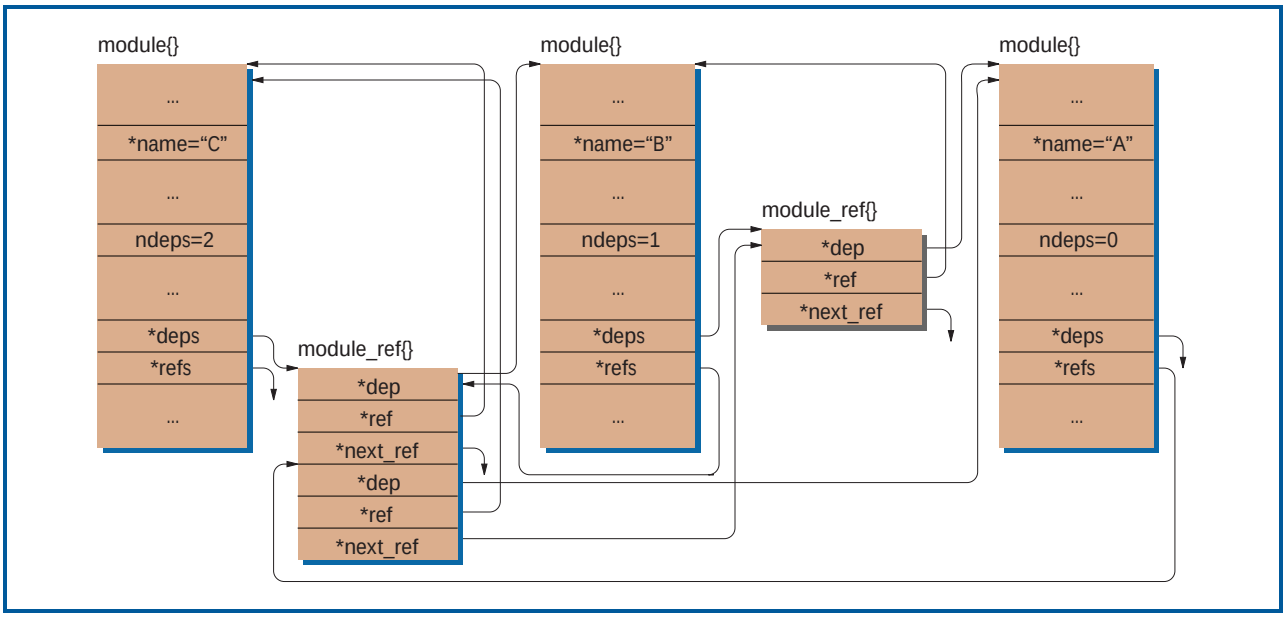

Figure 4. State after module C is inserted. C depends on both A and B.

The sources of the user space utilities `insmod` and `modprobe` are part of the `modutils` package, ftp://ftp.kernel.org/pub/linux/kernel/v2.1. Be sure to take the last version: it is backwards compatible with 2.0.x kernels, but also adds support for future 2.2 ones.

4. A. Rubini, *Linux Device Drivers*, O'Reilly & Associates, Sebastotol, Calif., 1998.
5. A. Rubini, "Dynamic Kernels: Modularize Device Drivers," *Linux J.*, Issue 23, Mar. 1996, http://www.ssc.com/lj/issue23/1219.html.
6. A. Rubini, "Dynamic Kernels: Discovery," *Linux J.*, Issue 24, Apr. 1996, http://www.ssc.com/lj/issue24/kk24.html.
7. G.v. Zezschwitz and A. Rubini, "The Devil's in the Details," *Linux J.*, Issue 25, May 1996, http://www.ssc.com/lj/issue25/kk25.html.
8. A. Rubini and G.v. Zezschwitz, "Dissecting Interrupts and Browsing DMA," *Linux J.*, Issue 26, June 1996, http://www.ssc.com/lj/issue26/interrupt.html.
9. D.A. Rusling, "The Linux Kernel," http://sunsite.unc.edu/linux/LDP/tlk/tlk.html.
10. R. Card, E. Dumas, and F. Mevel, *Programmation Linux 2.0 API systeme et fonctionnement du noyau*, Editions Eyrolles, Paris, 1997.
11. ELF specifications may be downloaded from ftp://sunsite.unc.edu/pub/Linux/GCC/ELF.doc.tar.gz.

About the Authors

Elena Apolinario Fernández de Sousa and **Juan-Mariano de Goyeneche** are undergraduate students at the Escuela Técnica Superior de Ingenieros de Telecomunicación, in the Technical University of Madrid (UPM), Spain. During the past few years they have been working for the Telematic Systems Department, on fellowships dealing with CSCW multimedia applications and multicast. When they have time, they enjoy diving into the Linux kernel sources. They have also collaborated with the GNU/Linux project writing some kernel patches, documentation, and articles.

Readers may contact de Sousa and Goyeneche at {jmseyas, elena}@selva.dit.upm.es.