Hacking the Linux 2.6 kernel, Part 2: Making your first hack

Kernel source, system calls, and kernel modules and patches

Skill Level: Introductory

Lina Mårtensson (linam@tyst.nu)
Freelance writer

Valerie Henson (val@nmt.edu)
Software Engineer
IBM

02 Aug 2005

In this second of a two-part series, discover the organization of the Linux kernel source, build an understanding of system calls, and craft your own kernel modules and patches.

Section 1. Before you start

Learn what these tutorials can teach you, and what you need to run the examples in them.

About this series

The capability of being modified is perhaps one of Linux's greatest strengths, and anyone who has dabbled with the source code has at least stood at the gates of the kingdom, if not opened them up and walked inside.

These two tutorials are intended to get you started. They are for anyone who knows a little bit of programming and who wants to contribute to the development of Linux, who feels that something is missing in the kernel and wants to fix that, or who just wants to find out how a real operating system works.

About this tutorial

This tutorial is a sequel to "Hacking the Linux 2.6 kernel, Part 1: Getting ready." Please read Part 1 before diving into Part 2.

We start where Part 1 left off by providing an overview of the kernel source. In this tutorial, we review where the various parts of the kernel are located in the source tree, what order they execute in, and how to go looking for a particular piece of code. We then explain system calls, teach you how to make your own modules, and finally instruct you on how to create, apply, and submit patches.

Prerequisites

To run the examples in this tutorial, you need a Linux box, root access on this Linux box (or a sympathetic admin), the ability to reboot this box several times a day, an installed compilation environment, and a way to get the kernel source.

The system prerequisites are covered in detail in Part 1 under "Requirement details." If you're not up on these details, you'll probably want to brush up before going on to the next section of this tutorial.

Section 2. Overview of the kernel source

The source tree

Let's start with the top-level directory of the Linux source tree, which is usually but not always in `/usr/src/linux-<version>`. We won't get too detailed, because the Linux source changes constantly, but we'll try to give you enough information to figure out where a certain driver or function is.

`Makefile`: This file is the top-level makefile for the whole source tree. It defines a lot of useful variables and rules, such as the default gcc compilation flags.

`Documentation/`: This directory contains a lot of useful (but often out of date) information about configuring the kernel, running with a ramdisk, and similar things. The help entries corresponding to different configuration options are not found here, though -- they're found in `Kconfig` files in each source directory.

`arch/`: All the architecture-specific code is in this directory and in the `include/asm-<arch>` directories. Each architecture has its own directory underneath this directory. For example, the code for a PowerPC-based computer would be found under `arch/ppc`. You will find low-level memory management, interrupt handling, early initialization, assembly routines, and much more in these directories.

`crypto/`: This is a cryptographic API for use by the kernel itself.

`drivers/`: As a general rule, code to run peripheral devices is found in subdirectories of this directory. This includes video drivers, network card drivers, low-level SCSI drivers, and other similar things. For example, most network card drivers are found in `drivers/net`. Some higher level code to glue all the drivers of one type together may or may not be included in the same directory as the low-level drivers themselves.

`fs/`: Both the generic filesystem code (known as the VFS, or Virtual File System) and the code for each different filesystem are found in this directory. One of the most commonly used filesystems in Linux is the ext2 filesystem; the code to read the ext2 format is found in `fs/ext2`. Not all of the filesystems compile or run; the more obscure filesystems are always a good candidate for someone looking for a kernel project.

`include/`: Most of the header files included at the beginning of a `.c` file are found in this directory. Architecture-specific include files are in `asm-<arch>`. Part of the kernel build process creates the symbolic link from `asm` to `asm-<arch>`, so that `#include <asm/file.h>` will get the proper file for that architecture without having to hardcode it into the `.c` file. The other directories contain non-architecture-specific header files. If a structure, constant, or variable is used in more than one `.c` file, it should be probably be in one of these header files.

`init/`: This directory contains the files `main.c`, code for creating *early userspace*, and other initialization code. `main.c` can be thought of as the kernel "glue." We'll talk more about `main.c` in the next section. Early userspace provides functionality that needs to be available while a Linux kernel is coming up, but that doesn't need to be run inside the kernel itself.

`ipc/`: *IPC* stands for *interprocess communication*. It contains the code for shared memory, semaphores, and other forms of IPC.

`kernel/`: Generic kernel-level code that doesn't fit anywhere else goes in here. The upper-level system-call code is here, along with the `printk()` code, the scheduler, signal-handling code, and much more. The files have informative names, so you can type `ls kernel/` and guess fairly accurately at what each file does.

`lib/`: Routines of generic usefulness to all kernel code are put in here. Common string operations, debugging routines, and command-line parsing code are all in here.

`mm/`: High-level memory-management code is in this directory. Virtual memory (VM) is implemented through these routines in conjunction with the low-level architecture-specific routines usually found in `arch/<arch>/mm/`. Early-boot memory management (needed before the memory subsystem is fully set up) is done here, as well as memory mapping of files, management of page caches, memory allocation, and swap out of pages in RAM (along with many other things).

`net/`: The high-level networking code is here. The low-level network drivers pass received packets up to and get packets to send from this level, which may pass the data to a user-level application, discard the data, or use it in-kernel, depending on the packet. The `net/core` directory contains code useful to most of the different network protocols, as do some of the files in the `net/` directory itself. Specific network protocols are implemented in subdirectories of `net/`. For example, IP (version 4) code is found in the directory `net/ipv4`.

`scripts/`: This directory contains scripts that are useful in building the kernel, but does not include any code that is incorporated into the kernel itself. The various configuration tools keep their files in here, for example.

`security/`: Code for different Linux security models can be found here, such as NSA Security-Enhanced Linux and socket and network security hooks, as well as other security options.

`sound/`: Drivers for sound cards and other sound related code is placed here.

`usr/`: This directory contains code that builds a cpio-format archive containing a root filesystem image which will be used for early userspace.

Where does it all come together?

The central connecting point of the whole Linux kernel is the file `init/main.c`. Each architecture executes some low-level set-up functions and then executes the function called `start_kernel` (which is found in `init/main.c`).

The order of execution of code looks something like this:

```
Architecture-specific set-up code (in arch/<arch>/*)
 |
 v
The function start_kernel() (in init/main.c)
 |
 v
The function init() (in init/main.c)
 |
 v
The user level "init" program
```

More details on the order of execution

In more detail, this is what happens:

- Architecture-specific set-up code that:

    - Unzips and moves the kernel code itself, if necessary
    - Initializes the hardware
        - This may include setting up low-level memory management
    - Transfers control to the function `start_kernel()`
- `start_kernel()` does, among other things:
    - Print out the kernel version and command line
    - Start output to the console
    - Enable interrupts
    - Calibrate the delay loop
    - Calls `rest_init()` which:
        - Starts a kernel thread to run the `init()` function
        - Enters the idle loop
- `init()`:
    - Starts the other processors (on SMP machines)
    - Starts the device subsystems
    - Mounts the root filesystem
    - Frees up unused kernel memory
    - Runs `/sbin/init` (or `/etc/init`, or...)

At this point, the userlevel `init` program is running; it will do things like start networking services and run `getty` (the login program) on your console(s).

You can figure out when a subsystem is initialized from `start_kernel()` or `init()` by putting in your own `printk`s and seeing when the `printk`s from that subsystem appear with regard to your own `printk`s. For example, if you wanted to find out when the ALSA sound system was initialized, put `printk`s at the beginning of `start_kernel()` and `init()` and look for where "Advanced Linux Sound Architecture [...]" is printed out relative to your `printk`s. (Part 2 offers help and tips for using the `printk()` function.)

Finding things in the kernel source tree

So, you want to start working on say, the USB driver. Where do you start looking for the USB code?

First, you can try a `find` command from the top-level kernel directory:

```
$ find . -name \*usb\*
```

This command will print out every filename that has the string "usb" somewhere in it.

Another thing you might try is looking for a unique string. This unique string can be the output of a `printk()`, the name of a file in `/proc`, or any other unique string that might be found in the source code for that driver. For example, USB prints out the message:

```
USB Universal Host Controller Interface driver v2.2
```

so you might try using a recursive grep to find the part of that `printk` that is not the version number:

```
$ grep -r "USB Universal Host Controller Interface driver" .
```

Another way you might try to find the USB source code is by looking in `/proc`. If you type `find /proc -name usb`, you might find that there is a directory named `/proc/bus/usb`. You might be able to find a unique string to grep for by reading the entries in that directory.

If all else fails, try descending into individual directories and listing the files or looking at the output of `ls -lR`. You may see a filename that looks related. But this should really be a last resort, something to be tried only after you have run many different find and grep commands.

Once you've found the source code you are interested in, you can start reading it. Reading and understanding Linux kernel code is another lesson in itself. Just remember that the more you read kernel code, the easier it gets. Have fun exploring the kernel!

Section 3. Understanding system calls

System calls

By now, you're probably looking around at device driver code and wondering, "How does the function `foo_read()` get called?" Or perhaps you're wondering, "When I type `cat /proc/cpuinfo`, how does the `cpuinfo()` function get called?"

Once the kernel has finished booting, the control flow changes from a comparatively straightforward "Which function is called next?" to being dependent on system calls, exceptions, and interrupts. In this tutorial, we'll talk about system calls.

What's a system call?

In the most literal sense, a system call (also called a "syscall") is an instruction similar to the "add" instruction or the "jump" instruction. At a higher level, a system call is the way a user-level program asks the operating system to do something for it. If you're writing a program and you need to read from a file, you use a system call to ask the operating system to read the file for you.

System calls in detail

Here's how a system call works. First, the user program sets up the arguments for the system call. One of the arguments is the system call number (more on that later). Note that all this is done automatically by library functions unless you are writing in assembly. After the arguments are all set up, the program executes the "system call" instruction. This instruction causes an exception: An event that causes the processor to jump to a new address and start executing the code there.

The instructions at the new address save your user program's state, figure out what system call you want, call the function in the kernel that implements that system call, restores your user program state, and returns control back to the user program. A system call is one way that the functions defined in a device driver end up being called.

That was the whirlwind tour of how a system call works. Next, we'll go into minute detail for those who are curious about exactly how the kernel does all this. Don't worry if you don't quite understand all of the details -- just remember that this is one way that a function in the kernel can end up being called -- no magic is involved. You can trace the control flow all the way through the kernel -- with difficulty sometimes, but you can do it.

A system call example: 1

This is a good place to start showing some code to go along with the theory. We'll follow the progress of a `read()` system call, starting from the moment the system call instruction is executed. The PowerPC architecture will be used as an example for the architecture specific part of the code. On the PowerPC, when you execute a system call, the processor jumps to the address `0xc00`. The code at that location is defined in the file `arch/ppc/kernel/head.S`. It looks something like this:

```
/* System call */
        . = 0xc00
SystemCall:
        EXCEPTION_PROLOG
        EXC_XFER_EE_LITE(0xc00, DoSyscall)

/* Single step - not used on 601 */
        EXCEPTION(0xd00, SingleStep, SingleStepException, EXC_XFER_STD)
        EXCEPTION(0xe00, Trap_0e, UnknownException, EXC_XFER_EE)
```

What this code does is save some state and call another function called `DoSyscall`. Here's a more detailed explanation (feel free to skip this part).

`EXCEPTION_PROLOG` is a macro that handles the switch from user to kernel space which requires things like saving the register state of the user process. `EXC_XFER_EE_LITE` is called with the address of this routine and the address of the function `DoSyscall`. Eventually, some state will be saved and `DoSyscall` will be called. The next two lines save two exception vectors on the addresses `0xd00` and `0xe00`.

`EXC_XFER_EE_LITE` looks like this:

```
#define EXC_XFER_EE_LITE(n, hdlr)       \
        EXC_XFER_TEMPLATE(n, hdlr, n+1, COPY_EE, transfer_to_handler, \
                          ret_from_except)
```

`EXC_XFER_TEMPLATE` is another macro and the code looks like this:

```
#define EXC_XFER_TEMPLATE(n, hdlr, trap, copyee, tfer, ret)     \
        li      r10,trap;                                       \
        stw     r10,TRAP(r11);                                  \
        li      r10,MSR_KERNEL;                                 \
        copyee(r10, r9);                                        \
        bl      tfer;                                           \
i##n:                                                           \
        .long   hdlr;                                           \
        .long   ret
```

`li` stands for *load immediate* which means that a constant value known at compile time is stored in a register. First, `trap` is loaded into the register `r10`. On the next line, that value is stored on the address given by `TRAP(r11)`. `TRAP(r11)` and the next two lines do some hardware-specific bit manipulation. After that we call the `tfer` function (the `transfer_to_handler` function) which does yet more housekeeping and then transfers control to `hdlr` (`DoSyscall`). Note that `transfer_to_handler` loads the address of the handler from the link register which is why you see `.long DoSyscall` instead of `bl DoSyscall`.

A system call example: 2

Now, let's look at `DoSyscall`. It's in the file `arch/ppc/kernel/entry.S`. Eventually, this function loads up the address of the syscall table and indexes into it using the system call number. The syscall table is what the OS uses to translate from a system call number to a particular system call.

The system call table is named sys_call_table and defined in arch/ppc/kernel/misc.S. The syscall table contains the addresses of the functions that implement each system call. For example, the read() system call function is named sys_read. The read() system call number is 3, so the address of sys_read() is in the fourth entry of the system call table (since we start numbering the system calls with 0). We read the data from the address sys_call_table + (3 * word_size) and we get the address of sys_read().

After DoSyscall has looked up the correct system call address, it transfers control to that system call. Let's look at where sys_read() is defined, in the file fs/read_write.c. This function finds the file struct associated with the fd number you passed to the read() function. That structure contains a pointer to the function that should be used to read data from that particular kind of file. After doing some checks, it calls that file-specific read() function in order to actually read the data from the file and then returns. This file-specific function is defined somewhere else -- the socket code, filesystem code, or device driver code, for example. This is one of the points at which a specific kernel subsystem finally interfaces with the rest of the kernel.

After our read function finishes, we return from the sys_read() back to DoSyscall() which switches control to ret_from_except (defined in arch/ppc/kernel/entry.S). This checks for tasks that might need to be done before switching back to user mode. If nothing else needs to be done, we fall through to the restore function which restores the user-process's state and returns control back to the user program.

There! Your read() call is done! If you're lucky, you even got your data back.

You can explore syscalls further by putting printks at strategic places. Be sure to limit the amount of output from these printks. For example, if you add a printk to sys_read() syscall, you should do something like this:

```
static int mycount = 0;

if (mycount < 10) {
    printk ("sys_read called\n");
    mycount++;
}
```

Have fun!

Section 4. Your first kernel module

Introduction

In this section, we'll write and load a simple kernel module. Writing your own module lets you write some standalone kernel code, learn how to use modules, and discover a few rules about how the kernel links together. **Note:** These instructions were written for the 2.6.x kernels and may not work with different kernel versions.

Does your kernel support modules?

For this section, your kernel must have been compiled with these options:

```
Loadable module support  --->

  [*] Enable loadable module support
  [*]   Module unloading
  [ ]   Module versioning support (EXPERIMENTAL)
  [*]   Automatic kernel module loading
```

If you compiled your kernel according to the instructions in the first tutorial, you should already have these options properly set. Otherwise, change these options, recompile the kernel, and boot into your new kernel.

A simple module skeleton

First, find the source that your current Linux kernel was compiled from. Change directory to `drivers/misc/` in your Linux source code directory. Now, copy and paste the following code into a file named `mymodule.c`:

```
#include <linux/module.h>
#include <linux/config.h>
#include <linux/init.h>

static int __init mymodule_init(void)
{
    printk ("My module worked!\n");
    return 0;
}

static void __exit mymodule_exit(void)
{
    printk ("Unloading my module.\n");
    return;
}

module_init(mymodule_init);
module_exit(mymodule_exit);

MODULE_LICENSE("GPL");
```

Save the file and edit the `Makefile` in the same directory. Add this line:

```
obj-m += mymodule.o
```

Compile your module:

```
# make -C <top directory of your kernel source> SUBDIRS=$PWD modules
```

Load the module with `insmod ./mymodule.ko` and check to see if your message printed out: `dmesg | tail`. You should see this at the end of the output:

```
My module worked!
```

Now remove the kernel module: `rmmod mymodule`. Check the output of dmesg again; you should see:

```
Unloading my module.
```

You just wrote and ran a new kernel module! Congratulations!

The module/kernel interface

Now, let's do some more interesting things with your module. One of the key things to realize is that modules can only "see" functions and variables that the kernel deliberately makes visible to the modules. First, let's try to do things the wrong way.

Edit the file `kernel/printk.c` and add this line after all the included files and near the other global variable declarations (but outside all functions):

```
int my_variable = 0;
```

Now recompile your kernel and reboot into your new kernel. Next, add this to the beginning of your module's `mymodule_init` function before the other code:

```
extern int my_variable;
printk ("my_variable is %d\n", my_variable);
my_variable++;
```

Save your changes and recompile your module:

```
# make -C <top directory of your kernel source> SUBDIRS=$PWD modules
```

And load the module (this will fail): `insmod ./mymodule.ko`. Loading your module should fail with the message:

```
insmod: error inserting './mymodule.ko': -1 Unknown symbol in module
```

What this is saying is that the kernel is not allowing modules to see that variable. When the module loads, it has to resolve all its external references like function names or variable names. If it can't find all of its unresolved names in the list of symbols that the kernel exports, then the module can't write to that variable or call that function. The variable `my_variable` has space allocated for it somewhere in the kernel, but the module can't figure out where.

To fix this, we're going to add `my_variable` to the list of symbols that the kernel exports. Many kernel directories have a file specifically for exporting symbols defined in that directory. Bring up the file `kernel/printk.c` again and add this line after the declaration of your variable:

```
EXPORT_SYMBOL(my_variable);
```

Recompile and reboot into your new kernel. Now try to load your module again: `insmod ./mymodule.ko`. This time, when you check dmesg, you should see:

```
my_variable is 0
My module worked!
```

Reload your module:

```
# rmmod mymodule && insmod ./mymodule.ko
```

Now you should see:

```
Unloading my module.
my_variable is 1
My module worked!
```

Each time you reload the module, `my_variable` should increase by one. You are reading and writing to a variable which is defined in the main kernel. Your module can access any variable or function in the main kernel, as long as it is explicitly exported via the `EXPORT_SYMBOL()` declaration. For example, the function `printk()` is defined in the kernel and exported in the file `kernel/printk.c`.

A simple loadable kernel module is a fun way to explore the kernel. For example, you can use a module to turn a `printk` on or off by defining a variable `do_print` in the kernel which is initially set to 0. Then make all your `printk`s dependent on "`do_print`":

```
if (do_print) {
   printk ("Big long obnoxious message\n");
}
```

And turn on `do_print` only when your module is loaded.

Module parameters

It is possible to pass arguments to your module when loading it. To load a module with module parameters, write:

```
insmod module.ko [param1=value param2=value ...]
```

To use the values from these parameters, declare variables to save them in in your module and use the macros `MODULE_PARM(variable, type)` and `MODULE_PARM_DESC(variable, description)` somewhere outside all functions to populate them. The `type` argument should be a string in the format `[min[-max]]{b,h,i,l,s}` where min and max delimit the length of an array. If both are omitted, the default is 1. The final character is a type specifier:

```
b      byte
h      short
i      int
l      long
s      string
```

You can add any description you like in the `description` field of `MODULE_PARM_DESC`.

Writing a module that uses interrupts

Now we're going to write a module that has a function that is called when the kernel receives an interrupt on a certain IRQ. First, copy the file `mymodule.c` to `myirqtest.c` and remove the contents of the functions with the exception of the return statements. Open `myirqtest.c` in your editor and replace the occurrences of "mymodule" with "myirqtest" to change the function names. Also remove the printks. To be able to use interrupts, add the line:

```
#include <linux/interrupt.h>
```

at the top of the file.

Use cat /proc/interrupts to find out what interrupts are in use. The first column tells you which interrupt number is used, the second how many times there have been interrupts on that IRQ since your computer was last booted, and the third which devices that use this IRQ. In this example, we will look at interrupts from a network interface and use two module parameters interface and irq to tell which interface and IRQ line that we want to use.

To take care of the module parameters, declare two variables to put them in and use MODULE_PARM and MODULE_PARM_DESC to catch the parameters. This code should be put somewhere outside all functions:

```
static int irq;
static char *interface;

MODULE_PARM(interface, "s");
MODULE_PARM_DESC(interface, "A network interface");
MODULE_PARM(irq, "i");
MODULE_PARM_DESC(irq, "The IRQ of the network interface");
```

The function request_irq() adds your function to the list of handlers for a selected IRQ line which you can use to print out a message each time you receive an interrupt on that line. Now, we need to request the IRQ for the network device in the function myirqtest_init. request_irq is defined as follows:

```
int request_irq(unsigned int irq,
   void (*handler)(int, void *, struct pt_regs *),
   unsigned long irqflags,
   const char *devname,
   void *dev_id);
```

irq is the interrupt number. We will use the value we received from the module parameter. handler is a pointer to the function that will handle the interrupt. We will name our handler function myinterrupt(). As the value for irqflags, we will use SA_SHIRQ which indicates that our handler supports sharing IRQ with other handlers. The devname is a short name for the device and is displayed in the /proc/interrupts list. We will use the value in the interface variable which we receive as a module parameter.

The dev_id parameter is the device ID. This parameter is often set to NULL, but it needs to be non-NULL if you want to share the IRQ so that the correct driver will be unhooked when the IRQ is freed using free_irq() later on. Since it's a void *, it can point to anything, but a common practice is to pass the driver's device structure. Here, we will use a pointer to our `irq` variable.

Upon success, `request_irq()` will return 0.

After writing the code, `myirqtest_init()` should look something like this:

```
static int __init myirqtest_init(void)
{
    if (request_irq(irq, &myinterrupt, SA_SHIRQ, interface, &irq)) {
        printk(KERN_ERR "myirqtest: cannot register IRQ %d\n", irq);
        return -EIO;
    }
    printk("Request on IRQ %d succeeded\n", irq);

    return 0;
}
```

If `request_irq()` doesn't return 0, something has gone wrong and the IRQ couldn't be registered, so we print out an error message and return with an error code.

Now, we also need to free the IRQ when we unload the module. This is done with `free_irq` which takes an interrupt number and a device ID as arguments. The interrupt number was saved in the `irq` variable and we used a pointer to this as the device ID, so all we need to do is to add this code in the beginning of `myirqtest_exit()`:

```
    free_irq(irq, &irq);
    printk("Freeing IRQ %d\n", irq);
```

All we have left to do now is to write the `myinterrupt()` handler function. The declaration of it has already been indirectly specified in on of the arguments to `request_irq()`: `void (*handler)(int, void *, struct pt_regs *)`. The first argument is the interrupt number, the second argument is the device ID that we used in `request_irq`, and the third argument holds a pointer to a structure containing the processor registers and state prior to servicing the interrupt.

Without looking at the processor registers, we won't know if an interrupt comes from our device or from some other device that shares the same IRQ. In this case, we will be satisfied with knowing that the interrupt was on the specified IRQ. When writing real drivers, it's important to check this and if the handler discovers that the interrupt was meant for another device, it should immediately return the value `IRQ_NONE` without handling the interrupt. If the interrupt was from our device and the handler was called correctly, `IRQ_HANDLED` should be returned. These operations are hardware specific and will not be covered here.

So, the `myinterrupt()` function will be called every time there's an interrupt on the specified IRQ. We will do a printout when this happens, but we want to limit the amount of output, so we'll do what we previously suggested and just do the printout on the first 10 interrupts.

We also need to return something from this function. Since this is not a real driver and we're just snooping the interrupts, we should return `IRQ_NONE`. By returning `IRQ_HANDLED` we would be saying that this is the real driver for the device and that no other driver needs to care about the interrupt (which is not true in this case).

Here's the resulting code for `myinterrupt()`:

```
static irqreturn_t myinterrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    static int mycount = 0;

    if (mycount < 10) {
        printk("Interrupt!\n");
        mycount++;
    }

    return IRQ_NONE;
}
```

And now we're done! Add the line:

```
obj-m += myirqtest.o
```

to the `Makefile` in this directory and compile your module with:

```
# make -C <top directory of your kernel source> SUBDIRS=$PWD modules
```

Now, insert your module (set the values of your parameters to something that will work on your system, see `cat /proc/interrupts`):

```
insmod myirqtest.ko interface=eth0 irq=9
```

Look at the printout from `dmesg`. It should look something like this:

```
Request on IRQ 9 succeeded
Interrupt!
Interrupt!
Interrupt!
Interrupt!
Interrupt!
```

There should be at most 10 lines of "Interrupt!" since we limited the number of printouts to that much. Now, unload the module:

```
rmmod myirqtest
```

The IRQ should now be freed from our handler. Check the output from dmesg. It should look like this:

```
Freeing IRQ 9
```

You have now made your own kernel module that uses interrupts! Play with your new kernel module -- modules are fun!

Section 5. Creating, applying, and submitting patches

Having a patch accepted

As a kernel developer, you'll spend a lot of time creating, applying, and submitting patches. Creating and applying patches can be tricky -- a lot of conventions must be learned and a lot of common mistakes avoided. Submitting a patch also takes some work.

At first, submitting patches might seem like the easiest part of kernel development. After all, it can't be as hard as fixing an Ethernet driver bug, right? Often it's easier to fix a kernel bug than to get a kernel patch accepted into the mainline kernel. Part of the reason is the sheer limitations of one person -- the kernel maintainer can only read and accept so many patches per release. But other reasons why patches are hard to get accepted are controversial changes, territoriality, personality conflicts, and apathy. And finally, whenever you submit a patch, you are putting your reputation and ego on the line and that's more than a little scary.

That being said, submitting a patch can be a lot of fun and very encouraging. Some kernel developers had parties to celebrate the first patch they wrote that was accepted into the mainline kernel. Knowing that you wrote some code that other people thought was good enough to include in the Linux kernel is a great feeling, so let's learn how to apply, create, and submit patches.

How patches work

A "patch" is a file that describes the differences between two versions of a file. The program `diff` compares the original file and the new file line-by-line and prints the differences to standard out in a specific format. The program `patch` can read the output of `diff` and apply those changes to another copy of the original file. (Note that the word "patch" refers both to the output of the `diff` command and to the command that applies the patch.) For example:

```
$ cat old/file.txt
This
is
a
simple
file.
$ cat new/file.txt
This
is
a
slightly more complex
file.
$ diff -uNr old new
diff -uNr old/file.txt new/file.txt
--- old/file.txt        Tue May 28 23:00:21 2002
+++ new/file.txt        Tue May 28 23:01:01 2002
@@ -1,5 +1,5 @@
 This
 is
 a
-simple
+slightly more complex
 file.
```

As you can see, the two files differ in only one line. The line from the first file listed on the command line is shown with a "-" in front of it, followed by the line from the second file on the command line is shown with a "+" in front of it. Intuitively, you are "subtracting" the line from the old file and "adding" the line from the new file. Remember, the old files always come first and the newer files second.

Now, let's apply the patch we just created. A patch updates the older version of the file to the newer version of the file, so we want to apply the patch to the older version of the file.

```
$ diff -uNr old new > patchfile
$ cd old
$ patch -p1 < ../patchfile
patching file file.txt
$ cat file.txt
This
is
a
slightly more complex
file.
```

After applying the output of the `diff` command using `patch`, the "old" file is now the same as the "new" file.

Applying patches

Next, we'll learn how to apply patches. One of the common reasons you'll need to apply a patch is in order to get a particular kernel version which isn't available as one big tarball downloadable from `ftp.kernel.org` -- or else to get an incremental patch so you don't have to download an entire new kernel when most of the kernel files are still the same.

The kernel patch naming and creation standards are not particularly simple. Say that you want to get the kernel `2.6.9-rc4`, for some reason, and you currently have the full kernel source for version `2.6.7`. You'll need to download the following patches to get from `2.6.7` to `2.6.9-rc4`:

`2.6.7` to `2.6.8`

`2.6.8` to `2.6.9-rc4`

Each prepatch (the patches that come between the major releases and are named `patch-2.6.x-rcN`, usually found in a directory on the ftp site called `testing`) is created by diffing against the previous major release. A common mistake is to download kernel version `2.6.9` and then attempt to apply the `2.6.9-rc4` prepatch. If you want kernel version `2.6.9-rc4`, you should download kernel `2.6.8` and then apply the `2.6.9-rc4` prepatch. This is because `2.6.9-rc4` is a predecessor of `2.6.9`, not the other way around. **Note:** The naming convention and location of kernel prepatches tends to change frequently. You may have to read the linux-kernel mailing list to find out where the very latest patches are being kept and what they are being named.

The official kernel patches are all made so that you can simply do the following:

```
cd <your linux source tree>
patch -p1 < ../patchfile
```

What the `-p1` option to the patch command says is to "strip the part of the pathname up through the first forward slash and then try to apply the patch to the file with the stripped down pathname."

If all this seems incredibly complex and annoying, you might want to try using Cogito. The end of this section carries a brief introduction to Cogito.

Creating a patch

The first thing to remember is to always keep an untouched, pristine version of the kernel source somewhere. Don't compile in it, don't edit any files in it, don't do anything to it -- just copy it to make your working copy of the source tree. The original kernel source should be in a directory named `linux.vanilla` or `linux.orig` and your working directory should be in the same directory as the original source. For example, if your pristine source is in `/usr/src/linux.vanilla`, your working source should be in `/usr/src/` also.

After you make your changes to your working copy, you'll create a patch using `diff`. Assuming that your working tree is named `linux.new`, you would run this command:

```
$ diff -upNr linux.vanilla linux.new > patchfile
```

All the differences between the original kernel source and your new kernel source are now in `patchfile`. **Note:** Do not ever create a patch with uneven directories, for example (DON'T do this):

```
$ diff -upNr linux.vanilla working/usb/thing1/linux > patchfile
```

This will not create a patch in the standard patch format and no one will bother trying out your patch since it's hard to apply.

Now that you've created a patch, read it! It's almost guaranteed that your patch includes files that you don't want as part of your patch, such as old editor back-up files, object files, or random junk data you created during development. To get rid of these files, you can tell diff to ignore certain files, you can delete the files, or you can hand-edit the diff. Be sure you understand the patch format before you hand-edit a patch or you can easily create a patch that won't apply. One useful command for getting rid of most of the extra files created during a kernel build is `make mrproper`.

But remember, this deletes your `.config` file and forces you to do a complete recompile of your kernel.

Also, make sure that your patch goes in the correct direction. Are your new lines the ones with "+" in front of them? And, make sure those are the changes you wanted to send. It's surprisingly easy to make a diff against the wrong source tree entirely.

After you think you've got a final version of the patch, apply it to a copy of your pristine source tree (don't ruin your only copy of the pristine source tree). If it doesn't apply without errors, redo the patch.

Once again, if this seems awfully complex, you may want to try Cogito.

What to think about before submitting your patch

After you've created a patch, you'll hopefully want to share it with other people. Ideally, you'll test the patch yourself, get other people to test it too, and have other people read the patch itself. In summary, you want your patch to be bug-free, well-written, and easy to apply.

Always compile and test your patches yourself. You'll see people posting "totally untested" patches to linux-kernel, but don't fall for it -- a totally untested patch is likely to be a useless patch. Kernel maintainers have more than once released a kernel which doesn't compile at all. No one is perfect -- always test your patches.

Be sure that your code fits in with the code around it and follows the kernel coding style conventions. See the file `Documentation/CodingStyle` for specific directions, although looking at other source files is often the best way to figure out what the current conventions are.

If it's difficult to apply your patch, it almost certainly won't be accepted. In addition to creating the patch with the proper level of directories, you need to create it against the kernel that is identical (or nearly so) to the kernel that other people will be applying your patch to. So if you want person XYZ to apply your patch, find out what version of the kernel person XYZ is using and try to get something as close to that as possible. Usually this is the latest vanilla kernel released by the kernel maintainer.

For example, if you have a patch against `2.6.9-rc2` and `2.6.9-rc4` is the latest version released, then you should recreate your patch against `2.6.9-rc4`. The easiest way to do this is to apply your patch from `2.6.9-rc2` to `2.6.9-rc4` and fix up any changes that occurred between the two versions, then rediff against `2.6.9-rc4`.

Who to submit your patch to

The answer to this question is "it depends." Subscribe to the Linux kernel mailing list and any list which is more specific to the area you are working on; you will begin to get an idea of who the appropriate person is.

Try to find the person most specifically involved in maintaining the part of the kernel you are changing. If you make a change to the foo driver in the bar subsystem and the foo driver has a maintainer, you should probably submit your patch to the foo maintainer and only to the bar subsystem maintainer if the foo maintainer is ignoring you.

The `MAINTAINERS` file in the top-level kernel source directory is frequently out of date, but often helpful anyway. No one will fault you if you send your patch to the person listed in the `MAINTAINERS` file. When in doubt, this is always the safest route to take.

Also, unless you have a reason not to do so, send your patch to the Linux kernel mailing list at linux-kernel@vger.kernel.org. Other developers than the maintainer might need to be informed about your change. They might also be of help by giving comments and suggestions.

Distributing your patch

Most patches are small enough to be included in an email. While some maintainers refuse to accept patches in attachments and some refuse MIME-encoded attachments, all maintainers will accept a patch that is included in the body of a text-only email. Make sure your mail client isn't mangling your patch -- if you aren't sure, email your patch to yourself and apply it to make sure other people will be able to apply it to. Most Linux mailing lists like patches with a meaningful English-language subject, prefixed with the string `[PATCH]` so that it's easy to find and read patches.

If your patch is too big to send by email (around 40 KB or larger), put it on a Web page or ftp site where other people can download it and put the URL in your email.

More guidelines on how to submit your patches can be found in the file `Documentation/SubmittingPatches` in the source tree.

Political considerations

If all that mattered is that your patch was well-formed, correct, and fixed a bug, submitting a patch would be a lot simpler. Instead, your patch needs to be tasteful, timely, interesting, and considerate of the maintainer's ego. Most of the time, a simple bugfix will be immediately accepted. Occasionally though, you'll run into bigger problems. The important thing to remember is that you can't work around the Linux maintainer system; you have to work through it.

Read a few threads on linux-kernel in which people tried to wheedle their patch into the kernel. If your patch isn't accepted, listen to what other people are saying about it and try to fix the problems with it. The most often rejected patch is the feature patch -- adding a new feature that is considered tasteless by the other maintainers. Don't waste your time trying to get that patch accepted, just maintain it separately. If enough people find the patch useful, you'll gain a reputation as being a useful kernel hacker among all the people who download and use your patch.

Sometimes, a maintainer just can't accept a patch because of his or her ego. When this happens, the only option is to maintain a better version of the code independently of the main kernel. Often, externally maintained code that proves to be better will replace the in-kernel code after a while -- which is one way to become a maintainer.

The alternative to diff and patch: Cogito

Cogito is currently being used by many kernel developers as a replacement for diff and patch. It simplifies a lot of kernel development tasks, such as updating to the latest version, creating patches, and applying patches.

To add a file, run:

```
$ cg-add file
```

To create a patch, run:

```
$ cg-diff > patchfile
```

To apply a patch, run:

```
$ cg-patch < patchfile
```

Section 6. Summary

In this tutorial series, you learned about different ways to get the kernel source, how to configure your own kernel, and how to boot it using a variety of bootloaders. You also got an overview of the kernel source, learned more about system calls, and learned how to write your own kernel modules and create patches.

You are now ready to explore the kernel source on you own, use the system calls, and build your own modules. See the Resources for links to topics for further exploration. For example, KernelTrap and Kernel Traffic offer news and discussions on kernel development.

Resources

Learn

- Part 1 of this series, "Hacking the Linux 2.6 kernel, Part 1: Getting ready," (developerWorks, July 2005) introduces ways to get the kernel source, how to configure your own kernel, and how to boot it using a variety of bootloaders.
- To learn more about how to use Cogito, take a look at the README file.
- The LILO mini-HOWTO and the GRUB manual are available online.
- "Inside the Linux kernel debugger" (developerWorks, June 2003) details KDB, the built-in kernel debugger in Linux, which allows you to trace the kernel execution and examine its memory and data structures.
- "Magic sys request" (developerWorks, April 2000) shows you how to recover from kernel meltdown.
- Find more resources for Linux developers in the developerWorks Linux zone.

Get products and technologies

- Get Cogito and the Linux kernel source at The Linux Kernel Archives.
- Order the no-charge SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2, Lotus, Rational, Tivoli, and WebSphere.
- Build your next development project on Linux with IBM trial software, available for download directly from developerWorks.

Discuss

- KernelNewbies.org has lots of resources for people who are new to hacking the kernel: an FAQ, an IRC channel, a mailing list, and a wiki.
- KernelTrap is a Web community devoted to sharing the latest in kernel development news.
- At Kernel Traffic you can find a newsletter that covers some of the discussion on the Linux kernel mailing list.
- Get involved in the developerWorks community by participating in developerWorks blogs.

About the authors

Lina Mårtensson
Lina Mårtensson is pursuing a M.Sc. in Computer Science and Engineering at Chalmers University of Technology, Sweden. Contact Lina at linam@tyst.nu.

Valerie Henson
Val Henson works for the Linux Technology Center at IBM. She has more than five years experience working on the Linux and Solaris operating systems, including a year as a maintainer of part of the PowerPC Linux kernel tree. Contact Val at val@nmt.edu.