Loadable Kernel Module

CSCI 3753: Operating Systems
Fall 2017

1. Introduction

If you want to add code to a Linux kernel, the most basic way to do that is to add some source files to the kernel source tree and recompile the kernel. This is what you did in your programming assignment one.

But you can also add code to the Linux kernel while it is running. A chunk of code that you add in this way is called a *loadable kernel module*. These modules can do lots of things, but they typically are one of three things: 1) device drivers; 2) filesystem drivers; 3) system calls. The kernel isolates certain functions, including these, especially well so they don't have to be intricately wired into the rest of the kernel. LKMs (when loaded) are very much part of the kernel. The correct term for the part of the kernel that is bound into the image that you boot, i.e. all of the kernel *except* the LKMs, is "base kernel." LKMs communicate with the base kernel.

LKMs have several advantages: (1) You don't have to rebuild your kernel; (2) LKMs help you diagnose system problems. A bug in a device driver which is bound into the kernel can stop your system from booting at all; (3) LKMs can save you memory, because you have to have them loaded only when you're actually using them; and (4) LKMs are much faster to maintain and debug.

There is a tendency to think of LKMs like user space programs. They do share a lot of their properties, but LKMs are definitely not user space programs. They are part of the kernel. As such, they have free run of the system and can easily crash it.

2. LKM Utilities

insmod: Insert an LKM into the kernel.
rmmod: Remove an LKM from the kernel.
depmod: Determine interdependencies between LKMs.
kerneld: Kerneld daemon program
ksyms: Display symbols that are exported by the kernel for use by new LKMs.
lsmod: List currently loaded LKMs.
modinfo: Display contents of .modinfo section in an LKM object file.
modprobe: Insert/remove an LKM or set of LKMs intelligently. e.g., if you must load A before loading B, modprobe will automatically load A when you tell it to load B.

We will discuss insmod in more detail here. Please check man pages and Internet for details of other utilities.

insmod

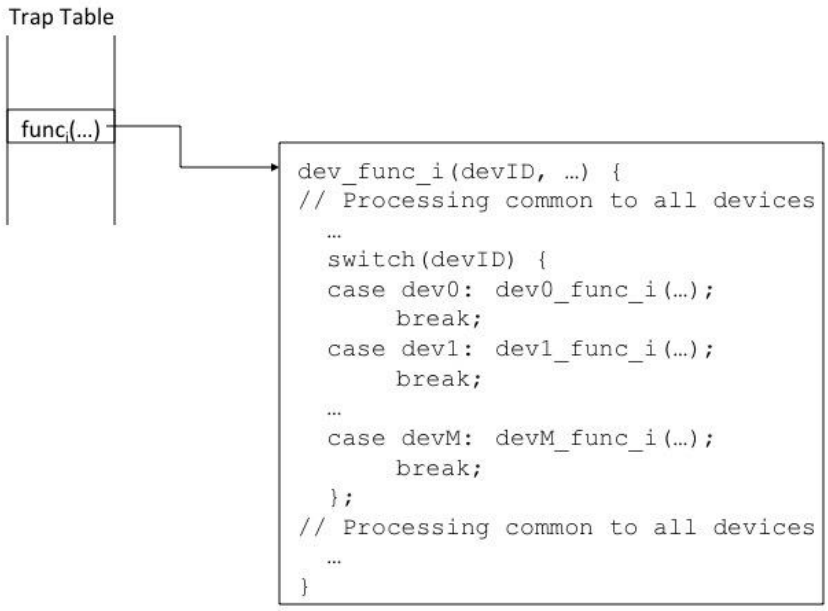
Example: *sudo insmod helloworld.ko*

insmod makes an *init_module* system call to load the LKM into kernel memory. The init_module system call invokes the LKM's initialization routine (also named *init_module*) right after it loads the LKM. **insmod** passes to init_module the address of the subroutine in the LKM named init_module as its initialization routine.

The LKM author sets up init_module to call a kernel function that registers the subroutines that the LKM contains. For example, a character device driver's init_module subroutine might call the kernel's *register_chrdev* subroutine, passing the major and minor number of the device it intends to drive and the address of its own "open", "close", "read", write" etc routines among the arguments. register_chrdev records in base kernel tables that when the kernel wants to open/close/read/write/… that particular device, it should call the open/close/read/write/… routine in our LKM.

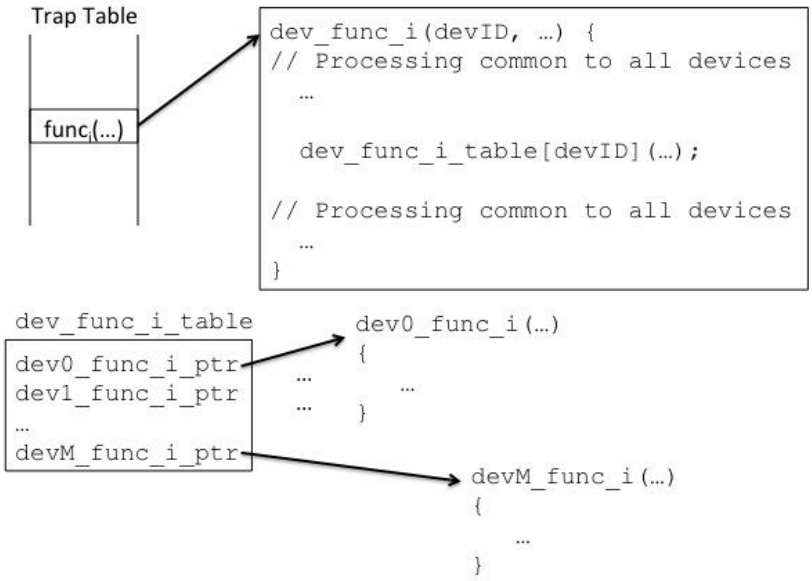
3. Adding a New Device Driver as an LKM

A traditional device driver code in the kernel is as follows:



So, to add a new device driver, a new case statement needs to be added under switch statement and then the kernel needs to be rebuilt.

For a kernel that allows loadable kernel module, a new level of indirection is added as shown in the figure below.



Here *dev_func_i_table* is a table of function pointers with each entry pointing to the device driver function (func_i) of the corresponding device. There is a separate such table for each device driver function (open, close, read, …). To add a new driver, a new entry is created in these tables pointing to the driver functions of the new device. e.g. if the new device is a character device, the LKM code will be as follows:

```
int init_module (…)
{
   …
  register_chrdev (…);
  …
}

int cleanup_module (…)
{
  …
}

int open (…)
{
  …
}

int read (…)
{
  …
}

…
```

The *register_chrdev* function (provided by the kernel) adds new entries in each dev_func_i_table pointing to the open, read, write, … functions.

4. How to Create an LKM

1. Install and prepare module-assistant.
2. Create hello.c and the Makefile.
3. Compile the code.
4. Insert the compiled module into the running kernel.
5. Remove the module when you're finished.

Debian and Ubuntu both provide module-assistant, a convenient package that contains all you need to write your own LKM.

Install and configure it with the following command:

```
$ sudo -i
# apt-get install module-assistant
# m-a prepare
```

module-assistant doesn't do anything *too* fancy. It's just a front-end that manages kernel source packages. The following command would also install the packages we need:

```
$  sudo apt-get install build-essential linux-headers-$(uname -r)
```

Here is the C source code for the hello world LKM, hello.c:

```
// Defining __KERNEL__ and MODULE allows us to access kernel-level code not
usually available to userspace programs.
#undef __KERNEL__
#define __KERNEL__

#undef MODULE
#define MODULE

// Linux Kernel/LKM headers: module.h is needed by all modules and kernel.h is
needed for KERN_INFO.
#include <linux/module.h>    // included for all kernel modules
#include <linux/kernel.h>    // included for KERN_INFO
#include <linux/init.h>        // included for __init and __exit macros

static int __init hello_init(void)
{
    printk(KERN_INFO "Hello world!\n");
    return 0;    // Non-zero return means that the module couldn't be loaded.
}



static void __exit hello_cleanup(void)
{
```

```
    printk(KERN_INFO "Cleaning up module.\n");
}

module_init(hello_init);
module_exit(hello_cleanup);
```

Here is the source for Makefile:

```
obj-m := hello.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
$(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
$(MAKE) -C $(KDIR) M=$(PWD) clean
```

To compile the code, go to your project directory and type make:

```
$ make
make -C /lib/modules/3.0.0-17-generic/build M=/var/www/lkm modules
make[1]: Entering directory `/usr/src/linux-headers-3.0.0-17-generic'
CC [M] /var/www/lkm/hello.o
Building modules, stage 2.
MODPOST 1 modules
CC /var/www/lkm/hello.mod.o
LD [M] /var/www/lkm/hello.ko
```

To insert your LKM into the running kernel, use insmod.

```
$ sudo insmod hello.ko
```

To see the message, tail the syslog:

```
$ tail /var/log/syslog
<snip>
Apr 20 16:27:39 laptop kernel: [19486.347191] Hello world!
```

Use rmmod to remove the LKM:

```
$ sudo rmmod hello
```