Project 6 Linux Kernel Modules

Due: Mon., Dec. 5th 11:59pm

Electronic copy with screenshots of output.

Do Not Test on ADA!

In this project, you will learn how to create a kernel module and load it into the Linux kernel. Although you may use an editor to write C programs, you will have to use the terminal application to compile the programs, and you will have to enter commands on the command line to manage the modules in the kernel.

As you'll discover, the advantage of developing kernel modules is that it is a relatively easy method of interacting with the kernel, thus allowing you to write programs that directly invoke kernel functions. It is important for you to keep in mind that you are indeed writing kernel code that directly interacts with the kernel. That normally means that any errors in the code could crash the system! However, since you will be using a virtual machine, any failures will at worst only require rebooting the system.

Part I—Creating Kernel Modules
The first part of this project involves following a series of steps for creating and inserting a module into the Linux kernel. You can list all kernel modules that are currently loaded by entering the command

lsmod

This command will list the current kernel modules in three columns: name, size, and where the module is being used.
       The following program (named simple.c and available with the source code for this text) illustrates a very basic kernel module that prints appropriate messages when the kernel module is loaded and unloaded.

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
/* This function is called when the module is loaded. */
int simple_init(void)
{
        printk(KERN INFO "Loading Module\n");
        return 0;
}
/* This function is called when the module is removed. */
void simple exit(void)
{
        printk(KERN INFO "Removing Module\n");
}
/* Macros for registering module entry and exit points. */
module_init(simple init);
```

```
module_exit(simple exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Simple Module");
MODULE_AUTHOR("SGG");
```

The function simple_init() is the **module entry point**, which represents the function that is invoked when the module is loaded into the kernel. Similarly, the simple_exit() function is the **module exit point**—the function that is called when the module is removed from the kernel. The module entry point function must return an integer value, with 0 representing success and any other value representing failure. The module exit point function returns void. Neither the module entry point nor the module exit point is passed any parameters. The two following macros are used for registering the module entry and exit points with the kernel:

```
module_init()
module_exit()
```

Notice how both the module entry and exit point functions make calls to the printk() function. printk() is the kernel equivalent of printf(), yet its output is sent to a kernel log buffer whose contents can be read by the dmesg command. One difference between printf() and printk() is that printk() allows us to specify a priority flag whose values are given in the <linux/printk.h> include file. In this instance, the priority is KERN INFO, which is defined as an *informational* message.

The final lines—MODULE LICENSE(), MODULE DESCRIPTION(), and MODULE AUTHOR() —represent details regarding the software license, description of the module, and author. For our purposes, we do not depend on this information, but we include it because it is standard practice in developing kernel modules.

This kernel module simple.c is compiled using the Makefile accompanying the source code with this project. To compile the module, enter the following on the command line:

```
make
```

The compilation produces several files. The file simple.ko represents the compiled kernel module. The following step illustrates inserting this module into the Linux kernel.

Loading and Removing Kernel Modules

Kernel modules are loaded using the insmod command, which is run as follows:

```
sudo insmod simple.ko
```

To check whether the module has loaded, enter the lsmod command and search for the module simple. Recall that the module entry point is invoked when the module is inserted into the kernel. To check the contents of this message in the kernel log buffer, enter the command

```
dmesg
```

You should see the message "Loading Module."

Removing the kernel module involves invoking the rmmod command (notice that the .ko suffix is unnecessary):

```
sudo rmmod simple
```

Be sure to check with the dmesg command to ensure the module has been removed.
Because the kernel log buffer can fill up quickly, it often makes sense to clear the buffer periodically. This can be accomplished as follows:

```
sudo dmesg –c
```

Part I Assignment

Proceed through the steps described above to create the kernel module and to load and unload the module. Be sure to check the contents of the kernel log buffer using dmesg to ensure you have properly followed the steps.

Part II Assignment

In the module entry point, create a linked list containing five struct person elements.

```
struct person{
        int month;
        int day;
        int year;
        struct person *next;
}
```

Don't use the kernel data structure as described in Part II on page 99. You may consider using a global struct pointer, which can be accessed by both entry point and exit point. Traverse the linked list and output its contents to the kernel log buffer. Invoke the dmesg command to ensure the list is properly constructed once the kernel module has been loaded (Take a screenshot of the message from "simple").

In the module exit point, delete the elements from the linked list and return the free memory back to the kernel. Again, invoke the dmesg command to check that the list has been removed once the kernel module has been unloaded (Take a screenshot Take a screenshot of the message from "simple").