CS4540 – Operating Systems

Assignment 5. Linux Kernel Modules

Due December 4

In this project, you will learn how to create a kernel module and load it into the Linux kernel. The project can be completed using the Ubuntu Linux running on a virtual machine. 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 a Simple Kernel Module

The first part of this project involves following a series of steps for creating and inserting a module into the Linux kernel.

I.1. An Illustrative Program for a Simple Kernel Module

Program code. The following program (named `simple.c`) is used to illustrate creating a very basic kernel module; this module prints two simple messages: one when the kernel module is loaded, and one when the kernel module is 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 "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");
```

Program explanation. 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.

Compilation. 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.

I.2. 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 II—Creating a Kernel Module Using Kernel Data Structures

The second part of this project involves modifying the kernel module so that it uses the kernel linked-list data structure.

We explore using the circular, doubly linked list that is available to kernel developers. Much of what we discuss is available in the Linux source code— in this instance, the include file `<linux/list.h>`—and you should examine this file to understand the following steps (`list.h` should be in the directory: `/usr/src/linux-headers-3.13.0-32/include/linux`).

II.1. A Kernel `struct` and its Description

Initially, you must define a `struct` containing the elements that are to be inserted in the linked list. The following C `struct` defines birthdays:

```
struct birthday {
   int day;
   int month;
   int year;
   struct list_head list;
}
```

Notice the member `struct list_head list`. The `list_head` structure is defined in the include file `<linux/types.h>` (`types.h` should be in the same directory as `list.h`). Its intention is to embed the linked list within the nodes that comprise the list. This `list_head` structure is quite simple—it merely holds two members, `next` and `prev`, that point to the next and previous entries in the list. By embedding the linked list within the structure, Linux makes it possible to manage the data structure with a series of *macro* functions (defined in `list.h`).

II.2. Inserting Elements into the Kernel Linked List

We can declare a `list_head` object, which we use as a reference to the head of the list by using the `LIST_HEAD()` macro

```
static LIST_HEAD(birthday_list);
```

This macro defines and initializes the variable `birthday_list`, which is of type `struct list_head`.

We create and initialize instances of `struct birthday` as follows:

```
struct birthday *person;

person = kmalloc(sizeof(*person), GFP KERNEL);
person->day = 2;
person->month= 8;
person->year = 1995;
INIT_LIST_HEAD(&person->list);
```

The `kmalloc()` function is the kernel equivalent of the user-level `malloc()` function for allocating memory, except that kernel memory is being allocated. (The `GFP_KERNEL` flag indicates routine kernel memory allocation.) The macro `INIT_LIST_HEAD()` initializes the `list` member in `struct birthday`. We can then add this instance to the end of the linked list using the `list_add_tail()` macro:

```
list_add_tail(&person->list, &birthday list);
```

II.3. Traversing the Kernel Linked List

Traversing the list involves using the `list_for_each_entry()` macro (see `list.h`), which accepts three parameters:

- A pointer to the structure being iterated over
- A pointer to the head of the list being iterated over
- The name of the variable containing the `list_head` structure

The following code illustrates this macro:

```
struct birthday *ptr;

list_for_each_entry(ptr, &birthday_list, list) {
   /* on each iteration ptr points */
   /* to the next birthday struct */
}
```

II.4. Removing Elements from the Kernel Linked List

Careful memory management—which includes releasing memory to prevent *memory leaks*—is crucial when developing kernel-level code. Before your program terminates, it must remove all list elements (nodes).

Removing elements from the list involves using the `list_del()` macro, which is passed a pointer to `struct list_head`

```
list_del(struct list_head *element)
```

This removes `element` from the list while maintaining the structure of the remainder of the list.

Perhaps the simplest approach for removing all elements from a linked list is to remove each element as you traverse the list. The macro `list_for_each_entry_safe()` behaves much like `list_for_each_entry()` except that it is passed an additional argument that maintains the value of the next pointer of the item being deleted. (This is necessary for preserving the structure of the list.) The following code example illustrates this macro:

```
struct birthday *ptr, *next

list for_each_entry_safe(ptr, next, &birthday list, list) {
   /* on each iteration ptr points */
```

```
    /* to the next birthday struct */
    list_del(&ptr->list);
    kfree(ptr);
}
```

Notice that after deleting each element, we return memory that was previously allocated with `kmalloc()` back to the kernel with the call to `kfree()`. Make sure that you print contents of each list element before removing it (this will play the role of confirming—to some extent—of the removal).

Part III—Implement Simple Kernel Module Using Illustrative Program

Use the code from Section I.1 to create the kernel module and to load and unload the module (proceed through the steps described above).

Be sure to display the contents of the kernel log buffer (using `dmesg`) in your report for this assignment.

Part IV— Implement Linked List Kernel Module

Entry point. In the module entry point, create a linked list containing six `struct birthday` elements. 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.

Exit point. 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.

Part V—Hints

H1) You need to use `#include <linux/slab.h>` for `kmalloc()`

> If you are using `kmalloc` or `kzalloc()` for memory allocation you have to include
> `#include <linux/slab.h>`
> They are called […] *slab allocators* […].

[http://stackoverflow.com/questions/13656913/compile-error-kernel-module]

H2) Updating Package Lists and Installing the Linux Kernel Source in Ubuntu

Download the package lists from the repositories and "update" them to get information on the newest versions of packages and their dependencies [1]:

```
sudo apt-get update
```

Note: APT, or Advanced Package Tool is a free software user interface that works with core libraries to handle the installation and removal of software on the Debian GNU/Linux distribution and its variants. [http://en.wikipedia.org/wiki/Advanced_Packaging_Tool]

Fetch new versions of packages existing on the machine if APT knows about these new versions by way of apt-get update, and handle intelligently the dependencies, so it might remove obsolete packages or add new ones [1]:

```
sudo apt-get dist-upgrade
```

Install just the headers in Ubuntu [2]:

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

Install the entire Linux kernel source in Ubuntu [2]:

```
sudo apt-get install linux-source
```

References (cf. VI below):

[1] http://askubuntu.com/questions/222348/what-does-sudo-apt-get-update-do
[2] http://stackoverflow.com/questions/16919512/linux-module-h-no-such-file-or-directory

H3) Makefiles for Linux Kernel

Source: "The Linux Kernel Module Programming Guide" by Peter Jay Salzman, available at:
http://tldp.org/LDP/lkmpg/2.6/html/lkmpg.html

CRITICAL: Generally, 'make' is used to build a kernel module and not a bare cc. More on this:

> Kernel modules need to be compiled a bit differently from regular userspace apps. Former kernel versions required us to care much about these settings, which are usually stored in Makefiles. Although hierarchically organized, many redundant settings accumulated in sublevel Makefiles and made them large and rather difficult to maintain.
> Fortunately, there is a new way of doing these things, called `kbuild`, and the build process for external loadable modules is now fully integrated into the standard kernel build mechanism. To learn more on how to compile modules which are not part of the official kernel (such as all the examples you'll find in this guide), see file linux/Documentation/kbuild/modules.txt.

See the whole Makefile example there—*very* important!
See also the *Hello World* example in Section 2 -- at least Subsections 2.1 - 2.5.

Here is `Makefile` that you must use:

```
obj-m += ProjectA5.o

all:
    make -C /lib/modules/$(shell uname -r)/build M="$(PWD)" modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M="$(PWD)" clean
```

H4) Summary of Commands for Compiling and Loading Kernel Modules

Compile the module (using the above `Makefile`):

```
make
```

List loaded modules:

```
lsmod
```

See that the module 'projectA5' is not loaded into the Ubuntu kernel.

Load your module into the kernel:

```
sudo insmod ProjectA5.ko
```

Note that '`.ko`' denotes a kernel .o file.

List loaded modules again:

```
lsmod
```

See that the module '`ProjectA5`' is shown now (that is, that it was successfully loaded into the kernel).

See the contents of the message that your module '`ProjectA5`' sent to the kernel log buffer:

```
dmesg
```

H5) Expected Output of `dmesg`.

In the output produced by `dmesg`, you might see a long list of messages. Look for the expected output of your kernel module. It should similar to this (starting with '`Loading Module`' and with all the birthday dates from the program in the list of birthdays):

```
[ 8641.136737] Loading Module
[ 8641.136741] Birthday: Month 10 Day 22 Year 1994
[ 8641.136742] Birthday: Month 8 Day 17 Year 1995
[ 8641.136743] Birthday: Month 11 Day 2 Year 1992
[ 8641.136744] Birthday: Month 8 Day 12 Year 1991
[ 8641.136745] Birthday: Month 12 Day 31 Year 1992
[ 8641.136746] Birthday: Month 4 Day 2 Year 1993
{directory-path}$
```

Part VI—Additional Background Information

VI.1. Copy of Text from Reference [1] (cf. VI.1 above)

http://askubuntu.com/questions/222348/what-does-sudo-apt-get-update-do

[QUESTION:]

I am wondering what sudo apt-get update does? What does it update?

edited Aug 31 at 5:34 Pandya
asked Nov 27 '12 at 0:07 Elysium

[ANSWER:]

In a nutshell, apt-get update doesn't actually install new versions of software.

- `apt-get update` downloads the package lists from the repositories and "updates" them to get information on the newest versions of packages and their dependencies. It will do this for all repositories and PPAs. From http://linux.die.net/man/8/apt-get:

Used to re-synchronize the package index files from their sources. The indexes of available packages are fetched from the location(s) specified in /etc/apt/sources.list(5). An update should always be performed before an `upgrade` or `dist-upgrade`.

- `apt-get upgrade` will fetch new versions of packages existing on the machine if APT knows about these new versions by way of apt-get update.

  From http://linux.die.net/man/8/apt-get:
  Used to install the newest versions of all packages currently installed on the system from the sources enumerated in /etc/apt/sources.list(5). Packages currently installed with new versions available are retrieved and upgraded; under no circumstances are currently installed packages removed, nor are packages that are not already installed retrieved and installed. **New versions of currently installed packages that cannot be upgraded without changing the install status of another package will be left at their current version.** [Emphasis mine] An update must be performed first so that apt-get knows that new versions of packages are available.

- `apt-get dist-upgrade` will do the same job which is done by `apt-get upgrade`, plus it will also intelligently handle the dependencies, so it might remove obsolete packages or add new ones. See here: What is "`dist-upgrade`" and why does it upgrade more than "`upgrade`"?

  From http://linux.die.net/man/8/apt-get:
  In addition to performing the function of upgrade, this option also intelligently handles changing dependencies with new versions of packages; apt-get has a "smart" conflict resolution system, and it will attempt to upgrade the most important packages at the expense of less important ones, if necessary. The /etc/apt/sources.list(5) file contains a list of locations from which to retrieve desired package files. See also apt_preferences(5) for a mechanism for over-riding the general settings for individual packages.

You can combine commands with && as follows:

```
sudo apt-get update&&sudo apt-get install foo bar baz foo-dev foo-dbg
```

or to get newest versions possible as per version requirements of dependencies:

```
sudo apt-get update&&sudo apt-get dist-upgrade
```

You need `sudo` both times, but since `sudo` by default doesn't prompt you within 5 or so minutes since the last `sudo` operation, you will be prompted for your password only once (or not at all).

edited May 25 at 18:57
answered Nov 27 '12 at 0:17 hexafraction

VI.2. Copy of Text from Reference [2] (cf. VI.1 above)

http://stackoverflow.com/questions/16919512/linux-module-h-no-such-file-or-directory

[QUESTION:]

For my thesis I am creating a Manet using the protocol ARAN. To install the protocol I'm using this manual, but the first step, the creation of trace_route, I received errors such as:

– linux/module.h: No such file or directory
– linux/procs_Fs: No such file or directory

– linux/skbuff: No such file or directory

I searched the web and found out that the problem is in the headers, but I do not find the solution ...

P.S. I am using Ubuntu 10.04 LTS Kernel 2.6.33 recompiled

edited Jun 6 '13 at 19:15 Vilhelm Gray
asked Jun 4 '13 at 13:43 Peppe Cook

[ANSWER:]

You're missing the Linux kernel headers which allow you to compile code against the Linux kernel.

To install just the headers in Ubuntu:

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

To install the entire Linux kernel source in Ubuntu:

```
$ sudo apt-get install linux-source
```

Note that you should use the kernel headers that match the kernel you are running.

answered Jun 4 '13 at 17:32 Vilhelm Gray