Práctica 2

“Manejo de módulos en Linux”

Sistemas Operativos

Dra. Mireya Paredes López

Fecha: 4/03/2020

Fecha de entrega:

Durante la clase anterior, vimos que hay diferentes tipos de estructuras de sistemas operativos. Linux es considerado un sistema operativo modular por su sistema de carga de programas a través de *módulos*. El objetivo de esta práctica es familiarizarse con el sistema de manejo de módulos de linux.

Ustedes encontrarán más información acerca de módulos en el capítulo 2 del libro de ***Operating System Concepts (Silberschatz)*** para resolver los siguientes problemas de programación.

Entregables:

En el syllabus del curso vienen las especificaciones de lo entregables.

Linux Kernel Modules

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 Linux virtual machine that you should have already installed by now. Although you may use an editor to write these 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 -f simple
```

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.

**Make a list of the commands used for handling modules and its description, and add it into your lab report. Notice that in the exam these commands will be tested.*

Part II—Kernel Data Structures

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

During the course, we covered *various data structures* that are common in operating systems. ***The Linux kernel provides several of these structures***. Here, 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>` —Before we continue, take a look into this file as you proceed through the following steps.

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

```
struct student {
      int ID;
      int age;
      int year;
      char *name;
      struct list_head list;
}
```

Notice the member `struct list_head list`. The `list_head` structure is defined in the include file `<linux/types.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.

Inserting Elements into the 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(student_list);
```

This macro defines and initializes the variable `student_list`, which is of type `struct list_head`.
We create and initialize instances of struct student as follows:

```
struct student *person;

person = kmalloc(sizeof(*person), GFP_KERNEL);
person->ID = 0;
person->age = 0;
person->year = 0;
strcpy(person->name, "");
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 student. We can then add this instance to the end of the linked list using the `list_add_tail()` macro:

```
List_add_tail(&person->list, &student_list);
```

Traversing the Linked List

Traversing the list involves using the `list_for_each_entry()` Macro, 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 student *ptr;

list for each entry(ptr, &student_list, list) {
      /* on each iteration ptr points */
      /* to the next student struct */
}
```

Removing Elements from the Linked List

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 student *ptr, *next

list_for_each_entry_safe(ptr,next,&student_list,list) {
      /* on each iteration ptr points */
      /* to the next student 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()`**.** Careful memory management—which includes releasing memory to prevent ***memory leaks***—is crucial when developing kernel-level code.

Part II Assignment

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

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.