Assignment 4:
Linux Kernel Modules

Deadline: 30th August 2015, 9:00 pm

Goal:

In this assignment, you will learn how to create a kernel module and load it into the Linux kernel. This assignment is the programming project described in the chapter 2 of the 9th edition of the textbook.

Part 1: Creating Kernel Modules

The first part of this assignment 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) 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.

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 assignment. 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 1 Practice 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 2: Kernel Data Structures
The second part of this assignment involves modifying the kernel module so that it uses the kernel linked-list data structure.

The Linux kernel provides several kernel data-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> — and we recommend that you examine 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 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>. 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(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);
```

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 birthday *ptr;
```

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

Assignment Deliverables – Part 2 Assignment
In the module entry point, create a linked list containing five 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.

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.

Submission Instructions:

Please submit the following:

1. A report explaining the how you implemented the part 2 the assignment. Please mention the version of the kernel you used.

2. The source code of the module entry and exit points that you developed.

Combine the report and source code into a zip file. Name it as ***assgn4-<rollno>.zip***. Also all name all the files in the zip document as **(1)** ***assgn4_report-<rollno>.pdf*** for the report files **(2)**

assgn4_src-<rollno>.c for the source code files. Submit your assignment by 30th August 2015, 9:00 pm.

Reading Materials and other Useful Links:

1. Operating System Concepts, Eight Edition by Avi Silberschatz,Peter Baer Galvin,Greg Gagne, Chapter 2, Page 93-97.
2. www.tldp.org/LDP/lkmpg/2.6/lkmpg.pdf