اصغر عباس نژاد

9313017

1-تشریح پروژه

**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 curently 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.

Simple.c:

#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 themodule 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—Kernel Data Structures**

The second part of this project involves modifying the kernel module so that

it uses the kernel linked-list data structure.

In Section 1.10, 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>—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 releasingmemory to prevent ***memory***

***leaks***—is crucial when developing kernel-level code.

**Part II Assignment**

In themodule 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 themodule 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.

این پروژ شامل دو بخش می باشد به صورت مختصر به توضیح ان می پردازیم :

بخش 1-ایجاد ماژول های هسته

اولین بخش این پروژه شامل یک سری مراحل برای ایجاد و درج یک ماژول در هسته لینوکس میباشد.

لیست تمام ماژول های هسته :lsmod

در این قسمت از پروژه ماژول ساده simple.c را به هسته اضافه میکنیم.که کد آن در پیوست امده است.

تابع simple\_init()، نقطه ورودی ماژول است که معرف تابعی است که موقع بارگذاری ماژول در هسته احضار میشود ومقدار صفر را بر میگرداند که نشانه ای موفقیت امیز بودن ان است و هر مقدار دیگر نشانه خطا است.

و همچنین تابع simple\_exit() نقطه خروج ماژول است .(تابعی که موقع خروج ماژول فراخوانی میشود.)

این دو تابع توسط دو ماکروی module\_init() و module\_exit() صدا زده می شود.

ماژول هسته توسط Makefile کامپایل میشود .

برای کامپایل انها از دستور make در خط فرمان استفاده میشود.

برای درج ماژول از sudo insmod simple.ko استفاده میکنیم.

برای بررسی محتوای پیام در بافر سابقه هسته از دستور dmesg استفاده میشود.

برای حذف ماژول از دستور sudo rmmod simple استفاده میشود.

برای پاک کردن بافر هسته از دستور sudo dmesg –c نیز استفاده میشود.

تمرین این بخش میخواد که که این مراحل را به صورت عملی انجام دهیم .که نتایج و راه حل های همگی به صورت فایل پیوست در پوشه screenshot/simple قرار دارند.

بخش 2-ساختمان داده های هسته

این بخش میخواهیم که از ساختمان داده لیست پیوندی استفاده کنیم.که پیاده سازی ان در فایل linklist امده است.

2-تشریح راه حل در فایل ضمیمه امده است.

3-نتایج

نتیجه ای دریافتی از این پروژه جذابیت کار با هسته لینوکس.