National University of Computer and Emerging Sciences

Operating System Lab - 09

Lab Manual

Contents

Objective

The objective of this lab is to demonstrate how to compile, load, list and remove modules from/to kernel and introduce module parameters i.e. how to add parameter in you module, adding parameter to your modules, types of parameter and setting parameter from outside kernel module. To introduce kernel symbol table and /proc file system.

Kernel Modules

"Modules are pieces of code that can be loaded and unloaded into the kernel upon demand."

Linux kernel modules are shared object files with extension '.o' (for kernel version 2.4 or lower) or '.ko' (for new kernel version). They are chunk of kernel code and they compiled separately. They can be removed or inserted at any time. They extend the functionality of the kernel without the need to reboot the system. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system.

Why We Need Kernel Modules?

Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality. So modules there are various advantages of modules but the most common attraction of creating modules is that it saves kernel image size and it also save RAM and disk storage.

List of Currently Loaded Modules

There are two ways in which you can list currently loaded modules. These are as follow

- View the contents of '/proc/modules' file using cat
- Run 'lsmod' command

Both of the above ways tell you which modules are currently loaded. However, the output format of both is different. 'lsmod' command will print loaded modules in long list format and 'cat /proc/modules' would list all the loaded modules but readable format is not guaranteed.

Command to Insert or Remove Modules

Command	Description	Example
insmod	Inserts module in kernel	insmod <module_name>.ko
rmmod	Removes module from kernel	rmmod <module_name>

Let's Google:

modprobe does the similar work like insmod. What is the difference between two??

Writing Kernel Modules

Writing Your First Module and Insert it to the Kernel

Before we begin, lets first go come back as a root user as we will work inside kernel mode and normal users are not permitted to play with it! Type the below commands to create directories and files.

```
~/$ sudo -i
~/$ mkdir /hello
~/$ cd /hello
/hello$ touch hello.c ; touch Makefile
```

Now that are directory and files are created, we need to put content in it.
Copy this content into the 'hello.c' file

```
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */
gediint init_module(void) {
printk(KERN_INFO "hello [INFO] Hellow World! \n");
/*
* A non 0 return means init_module failed; module can't be loaded.
*/
return 0;
}
void cleanup_module(void) {
printk(KERN_INFO " hello [INFO] Goodbye World! \n ");
}
```

Points to Ponder:

Kernel modules must have at least two functions:

- "start" (initialization) function called init_module() which is called when the module is insmoded into the kernel
- "end" (cleanup) function called cleanup_module() which is called just before it is rmmoded.

Typically, init_module() either registers a handler for something with the kernel, or it replaces one of the kernel functions with its own code (usually code to do something and then call the original function). The cleanup_module() function is supposed to undo whatever init_module() did, so the module can be unloaded safely.

Now copy the below content to 'Makefile'

```
obj-m += hello.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
```

Once the file is created we need to compile it. To compile the code we use the make command.
/hello$ make

If it compiled successfully you would get the following output.

```
root@OSLAB-VM:/hello# make
make -C /lib/modules/4.10.0-28-generic/build M=/hello modules
make[1]: Entering directory '/usr/src/linux-headers-4.10.0-28-generic'
  CC [M]  /hello/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /hello/hello.mod.o
  LD [M]  /hello/hello.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.10.0-28-generic'
```

Now we insert the compiled module to the kernel module's list so that it starts executing

```
/hello$ insmod ./hello.ko
```

To verify that our module is added to the list, we can do this by using the following command

```
/hello$ lsmod | grep hello
```

This should give the following output

```
root@OSLAB-VM:/hello# insmod hello.ko
root@OSLAB-VM:/hello# lsmod | grep hello
hello                  16384  0
root@OSLAB-VM:/hello#
```

To see that the output from our module run 'dmesg | tail -1'

"dmesg - print or control the kernel ring buffer"

```
root@OSLAB-VM:/hello# dmesg | tail -1
[ 5879.144436] hello [INFO] Hellow World!
root@OSLAB-VM:/hello#
```

You can see that init_module function is called however, cleanup_module is called when you remove your module, hence init_module function will setup the module which may call a function that will run forever in a while loop and cleanup_module is called to terminate that function and to stop the execution of the module. To remove the module from the list, use the following command.

```
/hello$ rmmod hello
/hello$ dmesg | tail -2
```

This should generate the following output

```
root@OSLAB-VM:/hello# rmmod hello
root@OSLAB-VM:/hello# dmesg | tail -2
[ 5879.144436] hello [INFO] Hellow World!
[ 6636.666179] hello [INFO] Goodbye World!
root@OSLAB-VM:/hello#
```

With module_init() & module_exit() i.e. Init Macros

As of Linux 2.4, you can rename the init and cleanup functions of your modules; they no longer have to be called init_module() and cleanup_module() respectively. This is done with the module_init() and module_exit() macros. These macros are defined in linux/init.h. The only caveat is that your init and cleanup functions must be defined before calling the macros, otherwise you'll get compilation errors. Here's an example of this technique: The below file was tested and compiled with name hello-2.c

```
/*
 *  hello-2.c - Demonstrating the module_init() and module_exit() macros.
*  This is preferred over using init_module() and cleanup_module().
 */
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */
#include <linux/init.h>                 /* Needed for the macros */

static int __init hello_2_init(void) {
        printk(KERN_INFO "hello-2 [INFO] Hellow World \n");
        return 0;
}

static void __exit hello_2_exit(void) {
        printk(KERN_INFO " hello-2 [INFO] Goodbye World \n ");
}

module_init(hello_2_init);
module_exit(hello_2_exit);
```

```
obj-m += hello.o
obj-m += hello-2.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
```

With Licensing and Documentation

```
[  609.639475] hello: module license 'unspecified' taints kernel.
[  609.639477] Disabling lock debugging due to kernel taint
[  609.640063] hello-2 [INFO] Hellow World
```

In kernel 2.4 and later, a mechanism was devised to identify code licensed under the GPL so people can be warned that the code is non open-source. This is accomplished by the MODULE_LICENSE() macro. By setting the license to GPL, you can keep the warning from being printed. This license mechanism is defined and documented in ‘linux/module.h.

- MODULE_DESCRIPTION() is used to describe what the module does
- MODULE_AUTHOR() declares the module's author
- MODULE_SUPPORTED_DEVICE() declares what types of devices the module supports.

These macros are all defined in linux/module.h and aren't used by the kernel itself. They're simply for documentation and can be viewed by a tool like objdump.

The following code demonstrate licensing and documentation of code.

```
#include <linux/module.h>     /* Needed by all modules */
#include <linux/kernel.h>     /* Needed for KERN_INFO */
#include <linux/init.h>       /* Needed for the macros */
#define DRIVER_AUTHOR "Sumaiyah Zahid <www.sumaiyahzahid.com>"
#define DRIVER_DESC   "A sample module code to demonstrate licensing and
documentation."

static int __init init_hello_3(void) {
 printk(KERN_INFO " hello-3 [INFO] Hellow World \n ");  return 0;
}
static void __exit cleanup_hello_3(void)
{
        printk(KERN_INFO " hello-3 [INFO] Goodbye World \n ");
}
module_init(init_hello_3);
module_exit(cleanup_hello_3);

/*
        Get rid of taint message by declaring code as GPL.
 */
MODULE_LICENSE("GPL");
/*
        Or with defines, like this:
 */
MODULE_AUTHOR(DRIVER_AUTHOR); /* Who wrote this module? */
MODULE_DESCRIPTION(DRIVER_DESC);          /* What does this module do */

/*
This module uses /dev/testdevice.  The MODULE_SUPPORTED_DEVICE macro might be
used in the future to help automatic configuration of modules, but is currently unused other
than for documentation purposes.
 */
MODULE_SUPPORTED_DEVICE("testdevice");
```

The above code is written on a file name ‘hello-3.c’ and the Makefile modified as previous. Now compile and run the following command

```
/hello$ modinfo hello-3.ko
```

The above command would get the following output

```
root@OSLAB-VM:/hello# modinfo hello-3.ko
filename:       /hello/hello-3.ko
description:    A sample module code to demonstrate licensing and documentation.
author:         Sumaiyah Zahid <www.sumaiyahzahid.com>
license:        GPL
srcversion:     10C153C58AE5A4E36C3CF3C
depends:
name:           hello_3
vermagic:       4.13.0-32-generic SMP mod_unload
root@OSLAB-VM:/hello#
```

printk()

0	KERN_EMERG	Emergency condition, system is probably dead
1	KERN_ALERT	Some problem has occurred, immediate attention is needed
2	KERN_CRIT	A critical condition
3	KERN_ERR	An error has occurred
4	KERN_WARNING	A warning
5	KERN_NOTICE	Normal message to take note of
6	KERN_INFO	Some information
7	KERN_DEBUG	Debug information related to the program

Module Parameters

Previously, we studied modules without parameters that have a static output and no user or time interaction. These modules are no good since they have a very limited functionality. Modules are just like functions that need parameters i.e. user input to produce variation in their outputs.

Addition of variables to a module is a bit tricky, since now we are working in kernel space and we don't have access to the screen so there is no direct way to get user inputs for our modules. What we do is to make our parameters static so that they could stand outside the module, and we could address them from there.

Well, there are three types of symbols according to their visibility scope:

1) STATIC: visible only in their source file

2) EXTERNAL: visible only for built – in kernel source code

3) EXPORTED: visible for apparently all loadable kernel modules

After addition of some parameters your module code looks like the following:

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/jiffies.h>

static int answer = 42;
int mod7_external_param =91;
int mod7_exported_param = 73;

static int __init mod7_in(void)
{
        printk(KERN_INFO "module mod5 being loaded. \n");
        printk(KERN_INFO "Current jiffies are: %lu. \n",jiffies);
        printk("Initial value for answer is %d. \n", answer);
        return 0;
}
static void __exit mod7_out(void)
{
        printk(KERN_INFO "final value for answer is %d. \n",answer);
        printk(KERN_INFO "module mod5 being unloaded. \n");
}
module_init(mod7_in);
module_exit(mod7_out);

//module_param(answer, int, 0644);
module_param_named(mod7_intparam, answer, int, 65);
MODULE_PARM_DESC(answer, "testing module parameters");

EXPORT_SYMBOL(mod7_exported_param);

MODULE_AUTHOR("ABC");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("symbols exported and otherwise...");
```

One more thing that although sounds a bit weird about the modules is that by themselves, modules never know that they have any parameter, so they don't expect any output from outside the module that is command line. So you need to explicitly tell a module that it do have a parameter named this, and of type this with these user rights outside you (module). You can do this by:

```
module_param(variable_name, variable_type, user_rights)
```

as written in comments in above code. The user rights have the octal value representing owner, group and others, prefixed by a zero. The reason that the line is commented out is that the very next line to code does exactly the same job with an extension.

If you know that your variable will be listed and used among those thousands of kernel's variables, you need to use some way to make it look unique in the list for the sake of identification. So preferably you would use something like myMod_var_int to make it stand out of the crowd, sounds good, but… using this long name for a variable that is redundant in your code may lead to frequent mistakes in writing the variable name, resulting in long list of errors when you would run "make". The solution to this is, there should be some way that you could use a smaller alias of the variable name in module during programming and use a longer more expressive or descriptive name for addressing it from outside, this is exactly what module_param_named does. It renames the variable to a new name that is visible to outside world letting you use your short less descriptive name inside. The syntax is similar except for that the list of parameters to this function also includes the new name of the variable.

```
module_param_named(new_variable_name, old_variable_name, variable_type, user_rights)
```

There are two exported kernel symbols that we are using in our module. First one is jiffies which is defined in jiffies.h representing cpu instruction logical timer value, and the other is the function printk() which we are using for output in log file.

Remember two things can be exported, variables and functions and these are jointly referred to as symbols in Kernel programming terms.

Modifying Parameter Value

The value of these parameters can be set by any of the three ways:

1- **Set a value by default during writing your program**

This is the simplest and least flexible method among the three that is you initialize variables at the time you declare them. This value is constant and persists till end if neither changed by program itself nor modified from outside.

2- **Set a value at the time of module insertion**

In this method you simply write the assignment statement right after insmod module_name.ko as shown in following

```
insmod Lab9M1.ko my_param=93
```

3- **Set a value during execution of module**

There is a system directory that keeps records of all running modules along with their parameters. If allowed in user rights, you can modify it from there using echo command like below:

```
echo 27 > /sys/module/Lab9M1/parameters/mod7_intparam=27
```

The Symbol Table

As you may have studied in COAL, operating systems manage programs data in the form of memory segments; the kernel addresses the data from the segments using their offset addresses. It is difficult to remember the offsets for each function and data that occurs in user code. An easy way to use them in our program is to give them short names called mnemonics. These mnemonics are referred in kernel as symbols. Each program maintains a table to record the mapping of addresses to these mnemonics with their respective modules, known as SYMBOL TABLE. You can view it by using the following command:

COMMAND

nm mod7.ko

OUTPUT

```
00000045 r __UNIQUE_ID_answer1
0000003a r __UNIQUE_ID_author2
00000000 r __UNIQUE_ID_description4
0000002e r __UNIQUE_ID_license3
0000006b r __UNIQUE_ID_mod7_intparamtype0
00000086 r __UNIQUE_ID_srcversion1
000000b2 r __UNIQUE_ID_vermagic0
00000000 r ____versions aaf12845 A
__crc_mod7_exported_param 00000074 r
__kcrctab_mod7_exported_param
00000000 r __kstrtab_mod7_exported_param
0000006c R __ksymtab_mod7_exported_param
000000a9 r __module_depends
00000000 r __param_mod7_intparam
00000000 r __param_str_mod7_intparam
00000000 D __this_module
00000000 b a.17573
00000008 d answer
00000000 T cleanup_module
00000000 T init_module
         U jiffies
00000004 D mod7_exported_param
00000000 D mod7_external_param
00000000 t mod7_in
00000000 t mod7_out
         U param_ops_int
         U printk
```

Whereas kernel keeps a separate list of similar pattern that contains global data only. This is a long list of symbols from all modules. Let's say you want to check the symbols entry for a particular module, you would write:

COMMAND

grep mod7 /proc/kallsyms

OUTPUT

```
f8945000 t mod7_out             [mod7]
 f8945168 d answer [mod7] f89452f4
 b a.17573 [mod7]
 f894513c r __kstrtab_mod7_exported_param      [mod7]
 f8945058 r __kcrctab_mod7_exported_param      [mod7]
 f894512c r __param_mod7_intparam    [mod7]
 f8945150 r __param_str_mod7_intparam          [mod7]
 f8945180 d __this_module    [mod7] f8945000
 t cleanup_module  [mod7] f8945160 d
 mod7_external_param         [mod7] f8945164
 D mod7_exported_param     [mod7] f8945050
 r __ksymtab_mod7_exported_param               [mod7]
```

The symbols defined here are closely related to system.map file, that file defines the types of symbols shown above as a single letter before the symbol name. These are defined as follows:

- A for absolute.
- B or b for uninitialized data section (called BSS).
- D or d for initialized data section.
- G or g for initialized data section for small objects (global).
- i for sections specific to DLLs.
- N for debugging symbol.
- p for stack unwind section.
- R or r for read only data section.
- S or s for uninitialzed data section for small objects.
- T or t for text (code) section.
- U for undefined.
- V or v for weak object.
- W or w for weak objects which have not been tagged so.
- - for stabs symbol in an a.out object file.
- ? for 'symbol type unknown.'

As said before you can see in listing of kernel symbols the presence of mod7_external_param and mod7_exported_param having not only different types but also mod7_exported_param is also listed in kernel string table and kernel crc table. These two tables maintain data for exported symbols. This variable is also listed in kernel symbol table as shown in last line.

You may also note the type ‘r’ in front of param_mod7_intparam, since it is only visible to kernel but kernel itself is also unable to use it, so they are written with the type read only.

Implementation of Kernel Threads

It is often useful for the kernel to perform some operations in the background. The kernel accomplishes this via kernel threads—standard processes that exist solely in kernel-space. Indeed, a kernel thread can be created only by another kernel thread.

```
#include<linux/module.h>
#include<linux/kernel.h>
#include<linux/kthread.h>
#include<linux/sched.h>
#include<linux/time.h>
#include<linux/timer.h>

static struct task_struct *thread1;
int a=21;

int threadFnc(void *t){
 int i = 1;
 int x=*(int *)t;
for(i = 1; i <= 10; i++)
{
printk(KERN_INFO "%d X %d = %d\n", i,x,x*i);
printk(KERN_INFO "");
```

```
}
do_exit(0);
return 0;
}
int thread_init(void) {
char our_thread[8]="thread1";
printk(KERN_INFO "in init");
thread1 = kthread_create(threadFnc,&a,our_thread);
if((thread1)) {
printk(KERN_INFO "in if");
wake_up_process(thread1);
}
return 0;
}
void threadCleanup(void) {
printk(KERN_INFO "cleaning up kthread");
printk(KERN_INFO "");
kthread_stop(thread1);
}
MODULE_LICENSE("GPL");
module_init(thread_init);
module_exit(threadCleanup);
```

Lab Task:

- Implement all the above codes and submit screenshots of Kernel messages.
- Implement Kthread for printing 5 tables.

Reference

- http://tldp.org/LDP/lkmpg/2.6/html/