An Introduction to the Linux Kernel

NARENDRA KANGRALKAR, JUNE 10, 2015

This article provides an introduction to the Linux kernel, and demonstrates how to write and compile a module.
Have you ever wondered how a computer manages the most complex tasks with such efficiency and accuracy? The answer is, with the help of the operating system. It is the operating system that uses hardware resources efficiently to perform various tasks and ultimately makes life easier. At a high level, the OS can be divided into two parts—the first being the kernel and other is the utility programs. Various user space processes ask for system resources such as the CPU, storage, memory, network connectivity, etc, and the kernel services these requests. This column will explore loadable kernel modules in GNU/Linux.

The Linux kernel is monolithic, which means that the entire OS runs solely in supervisor mode. Though the kernel is a single process, it consists of various subsystems and each subsystem is responsible for performing certain tasks. Broadly, any kernel performs the following main tasks.

Process management: This subsystem handles the process' 'life-cycle'. It creates and destroys processes, allowing communication and data sharing between processes through inter-process communication (IPC). Additionally, with the help of the process scheduler, it schedules processes and enables resource sharing.

Memory management: This subsystem handles all memory related requests. Available memory is divided into chunks of a fixed size called 'pages', which are allocated or de-allocated to/from the process, on demand. With the help of the memory management unit (MMU), it maps the process' virtual address space to a physical address space and creates the illusion of a contiguous large address space.

File system: The GNU/Linux system is heavily dependent on the file system. In GNU/Linux, almost everything is a file. This subsystem handles all storage related requirements like the creation and deletion of files, compression and journaling of data, the organisation of data in a hierarchical manner, and so on. The Linux kernel supports all major file systems including MS Windows' NTFS.

Device control: Any computer system requires various devices. But to make the devices usable, there should be a device driver and this layer provides that functionality. There are various types of drivers present, like graphics drivers, a Bluetooth driver, audio/video drivers and so on.

Networking: Networking is one of the important aspects of any OS. It allows communication and data transfer between hosts. It collects, identifies and transmits network packets. Additionally, it also enables routing functionality.

Dynamically loadable kernel modules
We often install kernel updates and security patches to make sure our system is up-to-date. In case of MS Windows, a reboot is often required, but this is not always acceptable; for instance, the machine cannot be rebooted if is a production server. Wouldn't it be great if we could add or remove functionality to/from the kernel on-the-fly without a system reboot? The Linux kernel allows dynamic loading and unloading of kernel modules. Any piece of code that can be added to the kernel at runtime is called a 'kernel module'. Modules can be loaded or unloaded while the system is up and running without any interruption. A kernel module is an object code that can be dynamically linked to the running kernel using the 'insmod' command and can be unlinked using the 'rmmod' command.

A few useful utilities
GNU/Linux provides various user-space utilities that provide useful information about the kernel modules. Let us explore them.

lsmod: This command lists the currently loaded kernel *modules*. This is a very simple program which reads the */proc/modules* file and displays its contents in a formatted manner.
insmod: This is also a trivial program which inserts a module in the kernel. This command doesn't handle module dependencies.

rmmod: As the name suggests, this command is used to unload modules from the kernel. Unloading is done only if the current module is not in use. *rmmod* also supports the *-f* or *–-force* option, which can unload modules forcibly. But this option is extremely dangerous. There is a safer way to remove modules. With the *-w* or *–-wait* option, *rmmod* will isolate the module and wait until the module is no longer used.

modinfo: This command displays information about the module that was passed as a command-line argument. If the argument is not a filename, then it searches the */lib/modules/<version>* directory for modules. modinfo shows each attribute of the module in the field:value format.
Note: <version> is the kernel version. We can obtain it by executing the uname -r command.

dmesg: Any user-space program displays its output on the standard output stream, i.e., */dev/stdout* but the kernel uses a different methodology. The kernel appends its output to the ring buffer, and by using the 'dmesg' command, we can manage the contents of the ring buffer.
Preparing the system
Now it's time for action. Let's create a development environment. In this section, let's install all the required packages on an RPM-based GNU/Linux distro like CentOS and a Debian-based GNU/Linux distro like Ubuntu.
Installing CentOS
First install the *gcc* compiler by executing the following command as a root user:

```
[root]# yum -y install gcc
```

Then install the kernel development packages:

```
[root]# yum -y install kernel-devel
```

Finally, install the 'make' utility:

```
[root]# yum -y install make
```

Installing Ubuntu
First install the gcc compiler:

```
[mickey] sudo apt-get install gcc
```

After that, install kernel development packages:

```
[mickey] sudo apt-get install kernel-package
```

And, finally, install the 'make' utility:

```
[mickey] sudo apt-get install make
```

Our first kernel module
Our system is ready now. Let us write the first kernel module. Open your favourite text editor and save the file as *hello.c* with the following contents:

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

int init_module(void)
{
printk(KERN_INFO "Hello, World !!!\n");

return 0;
}

void cleanup_module(void)
{
printk(KERN_INFO "Exiting ...\n");
}

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Narendra Kangralkar.");
MODULE_DESCRIPTION("Hello world module.");
```

```
MODULE_VERSION("1.0");
```

Any module must have at least two functions. The first is initialisation and the second is the clean-up function. In our case, *init_module()* is the initialisation function and *cleanup_module()* is the clean-up function. The initialisation function is called as soon as the module is loaded and the clean-up function is called just before unloading the module. *MODULE_LICENSE* and other macros are self-explanatory.

There is a *printk()* function, the syntax of which is similar to the user-space *printf()* function. But unlike *printf()* , it doesn't print messages on a standard output stream; instead, it appends messages into the kernel's ring buffer.
Each *printk()* statement comes with a priority. In our example, we used the *KERN_INFO* priority. Please note that there is no comma (,) between 'KERN_INFO' and the format string. In the absence of explicit priority, *DEFAULT_MESSAGE_LOGLEVEL* priority will be used. The last statement in *init_module()* is *return 0* which indicates success.

The names of the initialisation and clean-up functions are *init_module()* and *cleanup_module()* respectively. But with the new kernel (>= 2.3.13) we can use any name for the initialisation and clean-up functions. These old names are still supported for backward compatibility. The kernel provides *module_init* and *module_exit* macros, which register initialisation and clean-up functions. Let us rewrite the same module with names of our own choice for initialisation and cleanup functions:

```
#include <linux/kernel.h>
#include <linux/module.h>
static int __init hello_init(void)
{
printk(KERN_INFO "Hello, World !!!\n");

return 0;
}

static void __exit hello_exit(void)
{
printk(KERN_INFO "Exiting ...\n");
}

module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Narendra Kangralkar.");
MODULE_DESCRIPTION("Hello world module.");
MODULE_VERSION("1.0");
```

Here, the *__init* and *__exit* keywords imply initialisation and clean-up functions, respectively.

Compiling and loading the module

Now, let us understand the module compilation procedure. To compile a kernel module, we are going to use the kernel's build system. Open your favourite text editor and write down the following compilation steps in it, before saving it as *Makefile*. Please note that the kernel modules *hello.c* and *Makefile* must exist in the same directory.

```
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
```

To build modules, kernel headers are required. The above *makefile* invokes the kernel's build system from the kernel's source and finally the kernel's *makefile* invokes our *Makefile* to compile the module. Now that we have everything to build our module, just execute the make command, and this will compile and create the kernel module named *hello.ko:*

```
[mickey] $ ls
hello.c Makefile

[mickey]$ make
make -C /lib/modules/2.6.32-358.el6.x86_64/build M=/home/mickey modules
make[1]: Entering directory `/usr/src/kernels/2.6.32-358.el6.x86_64'
CC [M] /home/mickey/hello.o
Building modules, stage 2.
MODPOST 1 modules
CC /home/mickey/hello.mod.o
LD [M] /home/mickey/hello.ko.unsigned
NO SIGN [M] /home/mickey/hello.ko
make[1]: Leaving directory `/usr/src/kernels/2.6.32-358.el6.x86_64'
```

```
[mickey]$ ls
hello.c hello.ko hello.ko.unsigned hello.mod.c hello.mod.o hello.o Makefile modules
```

We have now successfully compiled our first kernel module. Now, let us look at how to load and unload this module in the kernel. Please note that you must have super-user privileges to load/unload kernel modules. To load a module, switch to the super-user mode and execute the *insmod* command, as shown below:

```
[root]# insmod hello.ko
```

insmod has done its job successfully. But where is the output? It is appended to the kernel's ring buffer. So let's verify it by executing the *dmesg* command:

```
[root]# dmesg
Hello, World !!!
```

We can also check whether our module is loaded or not. For this purpose, let's use the lsmod command:

```
[root]# lsmod | grep hello
hello 859 0
```

To unload the module from the kernel, just execute the *rmmod* command as shown below and check the output of the *dmesg* command. Now, *dmesg* shows the message from the clean-up function:

```
[root]# rmmod hello

[root]# dmesg
Hello, World !!!
Exiting ...
```

In this module, we have used a couple of macros, which provide information about the module. The *modinfo* command displays this information in a nicely formatted fashion:

```
[mickey]$ modinfo hello.ko
filename: hello.ko
version: 1.0
description: Hello world module.
author: Narendra Kangralkar.
license: GPL
srcversion: 144DCA60AA8E0CFCC9899E3
depends:
vermagic: 2.6.32-358.el6.x86_64 SMP mod_unload modversions
```

Finding the PID of a process

Let us write one more kernel module to find out the Process ID (PID) of the current process. The kernel stores all process related information in the *task_struct* structure, which is defined in the *<linux/sched.h>* header file. It provides a *current* variable, which is a pointer to the current process. To find out the PID of a current process, just print the value of the *current->pid* variable. Given below is the complete working code *(pid.c):*

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

static int __init pid_init(void)
{
printk(KERN_INFO "pid = %d\n", current->pid);

return 0;
}

static void __exit pid_exit(void)
{
/* Don't do anything */
}

module_init(pid_init);
module_exit(pid_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Narendra Kangralkar.");
MODULE_DESCRIPTION("Kernel module to find PID.");
MODULE_VERSION("1.0");
```

The *Makefile* is almost the same as the first *makefile,* with a minor change in the object file's name:

```
obj-m += pid.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
```

Now compile and insert the module and check the output using the dmesg command:

```
[mickey]$ make

[root]# insmod pid.ko

[root]# dmesg
pid = 6730
```

A module that spans multiple files

So far we have explored how to compile a module from a single file. But in a large project, there are several source files for a single module and, sometimes, it is convenient to divide the module into multiple files. Let us understand the procedure of building a module that spans two files. Let's divide the initialization and cleanup functions from the *hello.c* file into two separate files, namely *startup.c* and *cleanup.c.*

Given below is the source code for *startup.c:*

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

static int __init hello_init(void)
{
printk(KERN_INFO "Function: %s from %s file\n", __func__, __FILE__);

return 0;
}

module_init(hello_init);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Narendra Kangralkar.");
MODULE_DESCRIPTION("Startup module.");
MODULE_VERSION("1.0");
```

And "*cleanup.c*" will look like this.

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

static void __exit hello_exit(void)
{
printk(KERN_INFO "Function %s from %s file\n", __func__, __FILE__);
}

module_exit(hello_exit);

MODULE_LICENSE("BSD");
MODULE_AUTHOR("Narendra Kangralkar.");
MODULE_DESCRIPTION("Cleanup module.");
MODULE_VERSION("1.1");
```

Now, here is the interesting part — *Makefile* for these modules:

```
obj-m += final.o
final-objs := startup.o cleanup.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
```

The *Makefile* is self-explanatory. Here, we are saying: "Build the final kernel object by using *startup.o* and *cleanup.o*."

Let us compile and test the module:

```
[mickey]$ ls
cleanup.c Makefile startup.c

[mickey]$ make
```

Then, let's display module information using the modinfo command:

```
[mickey]$ modinfo final.ko
filename: final.ko
version: 1.0
description: Startup module.
author: Narendra Kangralkar.
license: GPL
version: 1.1
description: Cleanup module.
author: Narendra Kangralkar.
license: BSD
srcversion: D808DB9E16AC40D04780E2F
depends:
vermagic: 2.6.32-358.el6.x86_64 SMP mod_unload modversions
```

Here, the *modinfo* command shows the version, description, licence and author-related information from each module. Let us load and unload the *final.ko* module and verify the output:

```
[mickey]$ su -
Password:

[root]# insmod final.ko

[root]# dmesg
Function: hello_init from /home/mickey/startup.c file

[root]# rmmod final

[root]# dmesg
Function: hello_init from /home/mickey/startup.c file
Function hello_exit from /home/mickey/cleanup.c file
```

Passing command-line arguments to the module

In user-space programs, we can easily manage command line arguments with *argc/ argv*. But to achieve the same functionality through modules, we have to put in more of an effort.

To achieve command-line handling in modules, we first need to declare global variables and use the *module_param()* macro, which is defined in the *<linux/moduleparam.h>* header file. There is also the *MODULE_PARM_DESC()* macro which provides descriptions about arguments. Without going into lengthy theoretical discussions, let us write the code:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/moduleparam.h>

static char *name = "Narendra Kangralkar";
static long roll_no = 1234;
static int total_subjects = 5;
static int marks[5] = {80, 75, 83, 95, 87};

module_param(name, charp, 0);
MODULE_PARM_DESC(name, "Name of the a student");

module_param(roll_no, long, 0);
MODULE_PARM_DESC(rool_no, "Roll number of a student");

module_param(total_subjects, int, 0);
MODULE_PARM_DESC(total_subjects, "Total number of subjects");

module_param_array(marks, int, &total_subjects, 0);
MODULE_PARM_DESC(marks, "Subjectwise marks of a student");

static int __init param_init(void)
{
static int i;

printk(KERN_INFO "Name : %s\n", name);
printk(KERN_INFO "Roll no : %ld\n", roll_no);
printk(KERN_INFO "Subjectwise marks ");

for (i = 0; i < total_subjects; ++i) {
printk(KERN_INFO "Subject-%d = %d\n", i + 1, marks[i]);
}

return 0;
}

static void __exit param_exit(void)
{
/* Don't do anything */
}

module_init(param_init);
module_exit(param_exit);
```

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Narendra Kangralkar.");
MODULE_DESCRIPTION("Module with command line arguments.");
MODULE_VERSION("1.0");
```

After compilation, first insert the module without any arguments, which display the default values of the variable. But after providing command-line arguments, default values will be overridden. The output below illustrates this:

```
[root]# insmod parameters.ko

[root]# dmesg
Name : Narendra Kangralkar
Roll no : 1234
Subjectwise marks
Subject-1 = 80
Subject-2 = 75
Subject-3 = 83
Subject-4 = 95
Subject-5 = 87

[root]# rmmod parameters
```

Now, let us reload module with command-line arguments and verify the output.

```
[root]# insmod ./parameters.ko name="Mickey" roll_no=1001 marks=10,20,30,40,50

[root]# dmesg
Name : Mickey
Roll no : 1001
Subjectwise marks
Subject-1 = 10
Subject-2 = 20
Subject-3 = 30
Subject-4 = 40
Subject-5 = 50
```

If you want to learn more about modules, the Linux kernel's source code is the best place to do so. You can download the latest source code from *https://www.kernel.org/.* Additionally, there are a few good books available in the market like 'Linux Kernel Development' (3rd Edition) by Robert Love and 'Linux Device Drivers' (3rd Edition). You can also download the free book from http://lwn.net/Kernel/LDD3/.