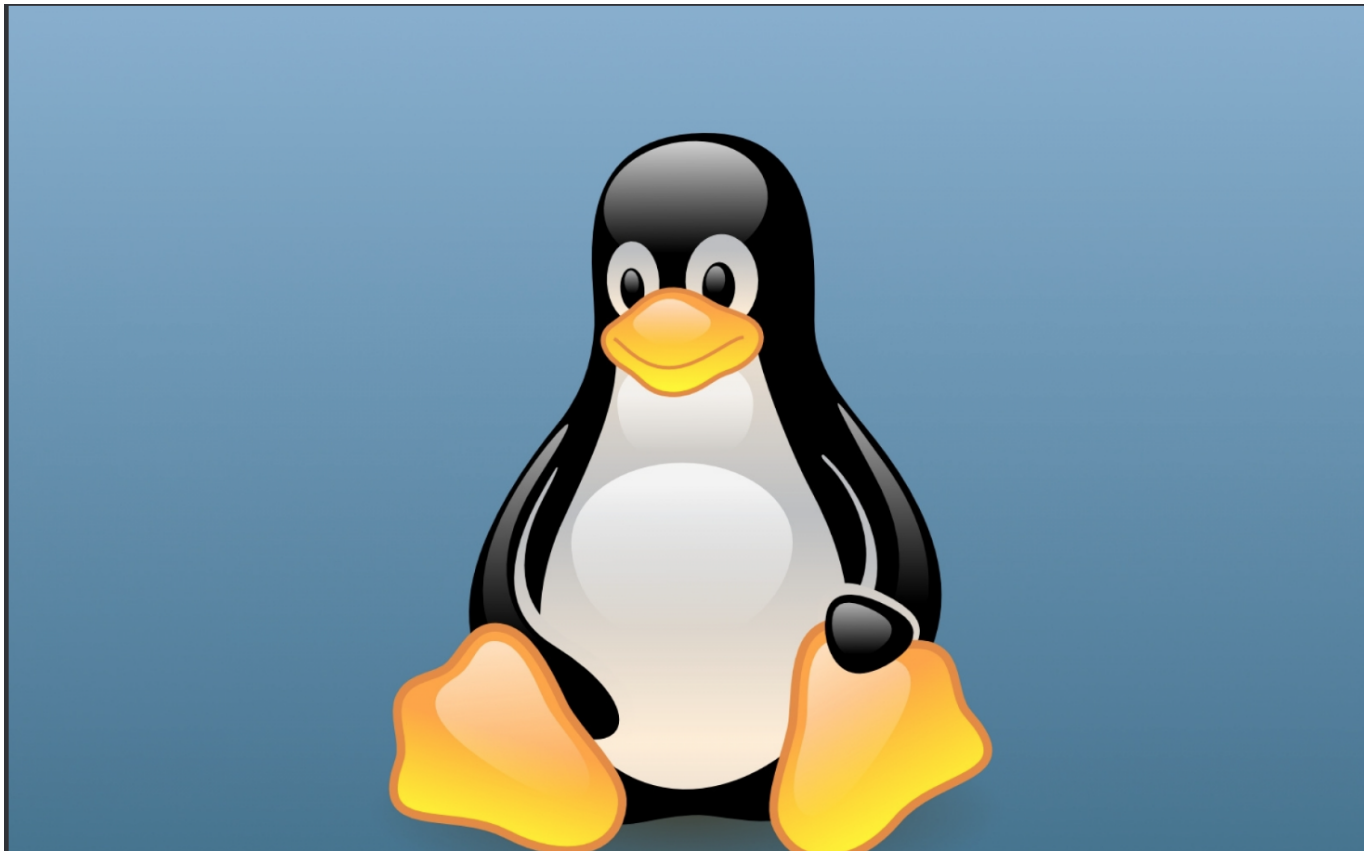
Writing a Simple Linux Kernel Module

Robert W. Oliver II Follow
Nov 30, 2017 · 9 min read



[Today's random Sourcerer profile: https://sourcerer.io/frankie]

Grabbing the Golden Ring-0

Linux provides a powerful and expansive API for applications, but sometimes that's not enough. Interacting with a piece of hardware or conducting operations that require accessed to privileged information in the system require a kernel module.

A Linux kernel module is a piece of compiled binary code that is inserted directly into the Linux kernel, running at ring 0, the lowest and least protected ring of execution in the x86–64 processor. Code here runs completely unchecked but operates at incredible speed and has access to everything in the system.

Not for Mere Mortals

Writing a Linux kernel module is not for the faint of heart. By altering the kernel, you risk data loss and system corruption. Kernel code doesn't have the usual safety net that regular Linux applications enjoy. If you have a fault, it will lock up the entire system.

To make matters worse, your issue may not become immediately apparent. Your module locking up immediately upon loading is probably the best-case scenario for failure. As you add more code to your module, you run the risk of introducing runaway loops and memory leaks. If you're not careful, these can continue to grow as your machine continues to run. Eventually important memory structures and even buffers can be overwritten.

Traditional application development paradigms can be largely discarded. Other than loading and unloading of your module, you'll be writing code that responds to system events rather than operates in a sequential pattern. With kernel development, you're writing APIs, not applications themselves.

You also have no access to the standard library. While the kernel provides some functions like printk (which serves as a replacement for printf) and kmalloc (which operates in a similar fashion to malloc), you are largely left to your own devices. Additionally, when your module unloads, you are responsible for completely cleaning up after yourself. There is no garbage collection.

Prerequisites

Before we get started, we need to make sure we have the correct tools for the job. Most importantly, you'll need a Linux machine. I know that comes as a complete surprise! While any Linux distribution will do, I am using Ubuntu 16.04 LTS in this example, so if you're using a different distribution you may need to slightly adjust your installation commands.

Secondly, you'll need either a separate physical machine or a virtual machine. I prefer to do my work in a virtual machine, but this is entirely up to you. I don't suggest using your primary machine because data loss can occur when you make a mistake. I say when, not if, because you undoubtedly will lock up your machine at least a few times during the process. Your latest code changes may still be in the write buffer when the kernel panics, so it's possible that your source files can become corrupted. Testing in a virtual machine eliminates this risk.

And finally, you'll need to know at least some C. The C++ runtime is far too large for the kernel, so writing bare metal C is essential. For interaction with hardware, knowing some assembly might be helpful.

Installing the Development Environment

On Ubuntu, we need to run:

```
apt-get install build-essential linux-headers-`uname -r`
```

This will install the essential development tools and the kernel headers necessary for this example.

The examples below assume you are running as a regular user and not root, but that you have sudo privileges. Sudo is mandatory for loading kernel modules, but we want to work outside of root whenever possible.

Getting Started

Let's start writing some code. Let's prepare our environment:

```
mkdir ~/src/lkm_example
cd ~/src/lkm_example
```

Fire up your favorite editor (in my case, this is vim) and create the file lkm_example.c with the following contents:

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

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Robert W. Oliver II");
MODULE_DESCRIPTION("A simple example Linux module.");
MODULE_VERSION("0.01");

static int __init lkm_example_init(void) {
 printk(KERN_INFO "Hello, World!\n");
 return 0;
}
```

```
static void __exit lkm_example_exit(void) {
 printk(KERN_INFO “Goodbye, World!\n”);
}

module_init(lkm_example_init);
module_exit(lkm_example_exit);
```

Now that we’ve constructed the simplest possible module, let’s example the important parts in detail:

· The “includes” cover the required header files necessary for Linux kernel development.

· MODULE_LICENSE can be set to a variety of values depending on the license of the module. To see a full list, run:
grep “MODULE_LICENSE” -B 27 /usr/src/linux-headers-`uname -r`/include/linux/module.h

· We define both the init (loading) and exit (unloading) functions as static and returning an int.

· Note the use of printk instead of printf. Also, printk doesn’t share the same parameters as printf. For example, the KERN_INFO, which is a flag to declare what priority of logging should be set for this line, is defined without a comma. The kernel sorts this out inside the printk function to save stack memory.

· At the end of the file, we call module_init and module_exit to tell the kernel which functions are or loading and unloading functions. This gives us the freedom to name the functions whatever we like.

We can’t compile this file yet, though. We need a Makefile. This basic example will work for now. Note that make is very picky about spaces and tabs, so ensure you use tab instead of space where appropriate.

```
obj-m += lkm_example.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
```

If we run “make”, it should compile your module successfully. The resulting file is “lkm_example.ko”. If you receive any errors, check that your quotation marks in the example source file are correct and not pasted accidentally as UTF-8 characters.

Now we can insert the module to test it. To do this, run:

```
sudo insmod lkm_example.ko
```

If all goes well, you won’t see a thing. The printk function doesn’t output to the console but rather the kernel log. To see that, we’ll need to run:

```
sudo dmesg
```

You should see the “Hello, World!” line prefixed by a timestamp. This means our kernel module loaded and successfully printed to the kernel log. We can also check to see if the module is still loaded:

```
lsmod | grep “lkm_example”
```

To remove the module, run:

```
sudo rmmod lkm_example
```

If you run dmesg again, you’ll see “Goodbye, World!” in the logs. You can also use lsmod again to confirm it was unloaded.

As you can see, this testing workflow is a bit tedious, so to automate this we can add:

```
test:
 sudo dmesg -C
 sudo insmod lkm_example.ko
 sudo rmmod lkm_example.ko
 dmesg
```

at the end of our Makefile and now run:

```
make test
```

to test our module and see the output of the kernel log without having to run separate commands.

Now we have a fully functional, yet completely trivial, kernel module!

A Bit More Interesting

Let's dive a bit deeper. While kernel modules can accomplish all sorts of tasks, interacting with applications is one of their most common uses.

Since applications are restricted from viewing the contents of kernel space memory, applications must use an API to communicate with them. While there are technically multiple ways to accomplish this, the most common is to create a device file.

You've likely interacted with device files before. Commands that use /dev/zero, /dev/null, or similar are interacting with a device named "zero" and "null" that return the expected values.

In our example, we'll return "Hello, World". While this isn't a particularly useful function to provide applications, it will nevertheless show the process of responding to an application via a device file.

Here's our complete listing:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <asm/uaccess.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Robert W. Oliver II");
MODULE_DESCRIPTION("A simple example Linux module.");
MODULE_VERSION("0.01");
```

```
#define DEVICE_NAME “lkm_example”
#define EXAMPLE_MSG “Hello, World!\n”
#define MSG_BUFFER_LEN 15

/* Prototypes for device functions */
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t,
loff_t *);

static int major_num;
static int device_open_count = 0;
static char msg_buffer[MSG_BUFFER_LEN];
static char *msg_ptr;

/* This structure points to all of the device functions */
static struct file_operations file_ops = {
 .read = device_read,
 .write = device_write,
 .open = device_open,
 .release = device_release
};

/* When a process reads from our device, this gets called. */
static ssize_t device_read(struct file *flip, char *buffer, size_t
len, loff_t *offset) {
 int bytes_read = 0;
 /* If we’re at the end, loop back to the beginning */
 if (*msg_ptr == 0) {
 msg_ptr = msg_buffer;
 }
 /* Put data in the buffer */
 while (len && *msg_ptr) {
 /* Buffer is in user data, not kernel, so you can’t just reference
 * with a pointer. The function put_user handles this for us */
 put_user(*(msg_ptr++), buffer++);
 len--;
 bytes_read++;
 }
 return bytes_read;
}

/* Called when a process tries to write to our device */
static ssize_t device_write(struct file *flip, const char *buffer,
size_t len, loff_t *offset) {
 /* This is a read-only device */
 printk(KERN_ALERT “This operation is not supported.\n”);
 return -EINVAL;
}

/* Called when a process opens our device */
static int device_open(struct inode *inode, struct file *file) {
 /* If device is open, return busy */
 if (device_open_count) {
```

```
 return -EBUSY;
 }
 device_open_count++;
 try_module_get(THIS_MODULE);
 return 0;
}

/* Called when a process closes our device */
static int device_release(struct inode *inode, struct file *file) {
 /* Decrement the open counter and usage count. Without this, the
module would not unload. */
 device_open_count--;
 module_put(THIS_MODULE);
 return 0;
}

static int __init lkm_example_init(void) {
 /* Fill buffer with our message */
 strncpy(msg_buffer, EXAMPLE_MSG, MSG_BUFFER_LEN);
 /* Set the msg_ptr to the buffer */
 msg_ptr = msg_buffer;
 /* Try to register character device */
 major_num = register_chrdev(0, "lkm_example", &file_ops);
 if (major_num < 0) {
 printk(KERN_ALERT "Could not register device: %d\n", major_num);
 return major_num;
 } else {
 printk(KERN_INFO "lkm_example module loaded with device major
number %d\n", major_num);
 return 0;
 }
}

static void __exit lkm_example_exit(void) {
 /* Remember — we have to clean up after ourselves. Unregister the
character device. */
 unregister_chrdev(major_num, DEVICE_NAME);
 printk(KERN_INFO "Goodbye, World!\n");
}

/* Register module functions */
module_init(lkm_example_init);
module_exit(lkm_example_exit);
```

Testing Our Enhanced Example

Now that our example does something more than simply print a message upon loading and unloading, we need a less restrictive test routine. Let's modify our Makefile to only load the module and not unload it.

```
obj-m += lkm_example.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

test:
  # We put a - in front of the rmmod command to tell make to ignore
  # an error in case the module isn't loaded.
  -sudo rmmod lkm_example
  # Clear the kernel log without echo
  sudo dmesg -C
  # Insert the module
  sudo insmod lkm_example.ko
  # Display the kernel log
  dmesg
```

Now when you run "make test", you'll see the output of the device's major number. In our example, this is automatically assigned by the kernel. However, you'll need this value to create the device.

Take the value you obtain from "make test" and use it to create a device file so that we can communicate with our kernel module from user space.

```
sudo mknod /dev/lkm_example c MAJOR 0
```

(in the above example, replace MAJOR with the value you obtain from "make test" or "dmesg")

The "c" in the mknod command tells mknod that we need a character device file created.

Now we can grab content from the device:

```
cat /dev/lkm_example
```

or even via the "dd" command:

```
dd if=/dev/lkm_example of=test bs=14 count=100
```

You can also access this device via applications. They don't have to be compiled applications — even Python, Ruby, and PHP scripts can access this data.

When we're done with the device, delete it and unload the module:

```
sudo rm /dev/lkm_example
sudo rmmod lkm_example
```

Conclusion

I hope you've enjoyed our romp through kernel land. Though the examples I've provided are basic, you can use this structure to construct your own module that does very complex tasks.

Just remember that you are completely on your own in kernel land. There are no backstops or second chances for your code. If you're quoting a project for a client, be sure to double, if not triple, the anticipated debugging time. Kernel code has to be as perfect as possible to ensure the integrity and reliability of the systems that will run it.

Follow me and other software engineers on Sourcerer Blog

Programming Linux Software Development Linux Kernel Software Engineering

About Help Legal