

Device Drivers, Part 2: Writing Your First Linux Driver in the Classroom

By **Anil Kumar Pugalia** - December 1, 2010



Sketching a driver

This article, which is part of the [series on Linux device drivers](#), deals with the concept of dynamically loading drivers, first writing a Linux driver, before building and then loading it.

Shweta and Pugs reached their classroom late, to find their professor already in the middle of a lecture. Shweta sheepishly asked for his permission to enter. An annoyed Professor Gopi responded, "Come on! You guys are late again; what is your excuse, today?"

Pugs hurriedly replied that they had been discussing the very topic for that day's class — device drivers in Linux. Pugs was more than happy when the professor said, "Good! Then explain about dynamic loading in Linux. If you get it right, the two of you are excused!" Pugs knew that one way to make his professor happy was to criticise Windows.

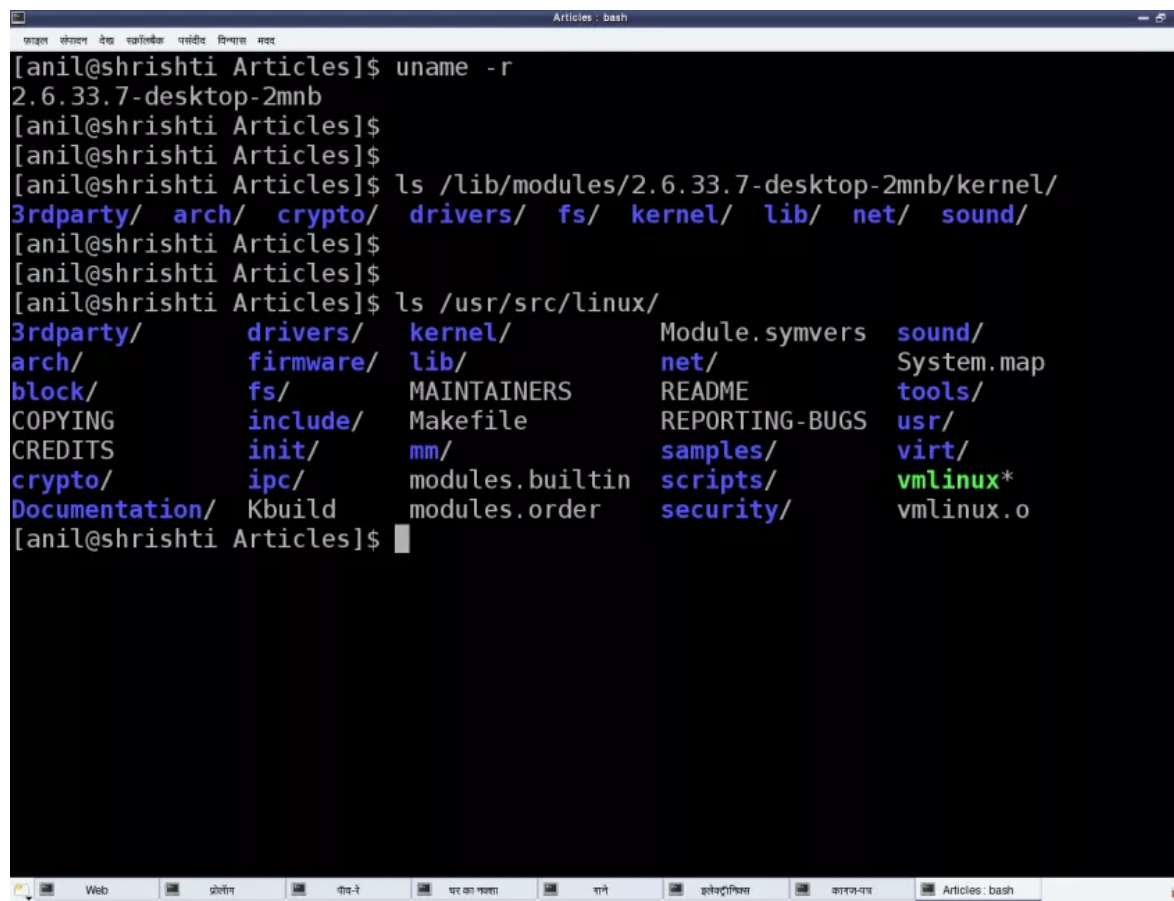
He explained, "As we know, a typical driver installation on Windows needs a reboot for it to get activated. That is really not acceptable; suppose we need to do it on a server? That's where Linux wins. In Linux, we can load or unload a driver on the fly, and it is active for use instantly after loading. Also, it is instantly disabled when unloaded. This is called dynamic loading and unloading of drivers in Linux."

This impressed the professor. "Okay! Take your seats, but make sure you are not late again." The professor continued to the class, "Now you already know what is meant by dynamic loading and unloading of drivers, so I'll show you how to do it, before we move on to write our first Linux driver."

Dynamically loading drivers

Get real-time push notifications

These dynamically loadable drivers are more commonly called modules and built into individual files with a `.ko` (kernel object) extension. Every Linux system has a standard place under the root of the file system (`/`) for all the pre-built modules. They are organised similar to the kernel source tree structure, under `/lib/modules/<kernel_version>/kernel`, where `<kernel_version>` would be the output of the command `uname -r` on the system, as shown in Figure 1.



```

[anil@shrishti Articles]$ uname -r
2.6.33.7-desktop-2mnb
[anil@shrishti Articles]$
[anil@shrishti Articles]$
[anil@shrishti Articles]$ ls /lib/modules/2.6.33.7-desktop-2mnb/kernel/
3rdparty/  arch/  crypto/  drivers/  fs/  kernel/  lib/  net/  sound/
[anil@shrishti Articles]$
[anil@shrishti Articles]$
[anil@shrishti Articles]$ ls /usr/src/linux/
3rdparty/  drivers/  kernel/  Module.symvers  sound/
arch/      firmware/ lib/      net/             System.map
block/     fs/       MAINTAINERS  README          tools/
COPYING    include/  Makefile    REPORTING-BUGS  usr/
CREDITS    init/     mm/         samples/        virt/
crypto/    ipc/      modules.builtin  scripts/       vmlinux*
Documentation/ Kbuild   modules.order  security/      vmlinux.o
[anil@shrishti Articles]$

```

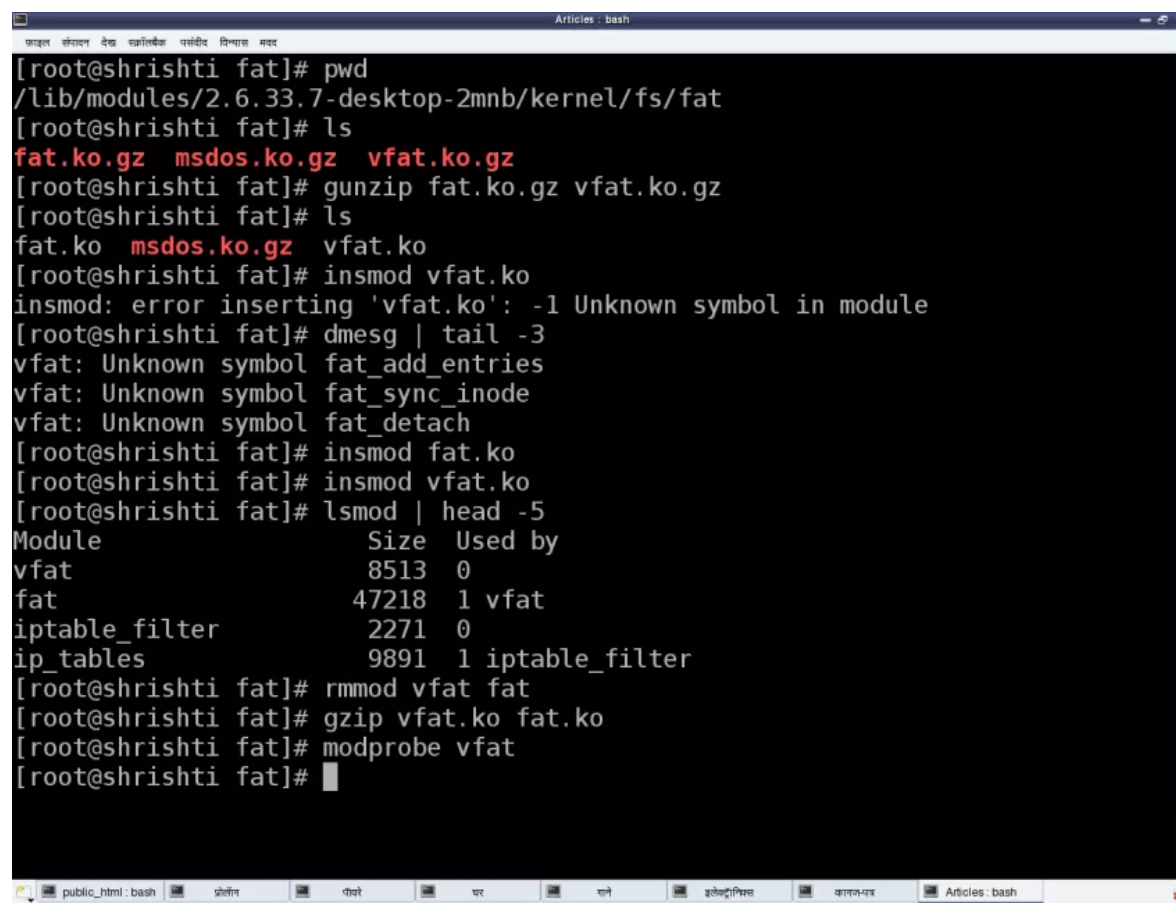
Figure 1: Linux pre-built modules

To dynamically load or unload a driver, use these commands, which reside in the `/sbin` directory, and must be executed with root privileges:

Get real-time push notifications

- `lsmod` — lists currently loaded modules
- `insmod <module_file>` — inserts/loads the specified module file
- `modprobe <module>` — inserts/loads the module, along with any dependencies
- `rmmod <module>` — removes/unloads the module

Let's look at the FAT filesystem-related drivers as an example. Figure 2 demonstrates this complete process of experimentation. The module files would be `fat.ko`, `vfat.ko`, etc., in the `fat` (`vfat` for older kernels) directory under `/lib/modules/`uname -r`/kernel/fs`. If they are in compressed `.gz` format, you need to uncompress them with `gunzip`, before you can `insmod` them.



```

Articles : bash
[root@shrishti fat]# pwd
/lib/modules/2.6.33.7-desktop-2mnb/kernel/fs/fat
[root@shrishti fat]# ls
fat.ko.gz  msdos.ko.gz  vfat.ko.gz
[root@shrishti fat]# gunzip fat.ko.gz vfat.ko.gz
[root@shrishti fat]# ls
fat.ko  msdos.ko  vfat.ko
[root@shrishti fat]# insmod vfat.ko
insmod: error inserting 'vfat.ko': -1 Unknown symbol in module
[root@shrishti fat]# dmesg | tail -3
vfat: Unknown symbol fat_add_entries
vfat: Unknown symbol fat_sync_inode
vfat: Unknown symbol fat_detach
[root@shrishti fat]# insmod fat.ko
[root@shrishti fat]# insmod vfat.ko
[root@shrishti fat]# lsmod | head -5
Module                Size  Used by
vfat                   8513   0
fat                   47218  1 vfat
iptables_filter       2271   0
ip_tables             9891  1 iptable_filter
[root@shrishti fat]# rmmod vfat fat
[root@shrishti fat]# gzip vfat.ko fat.ko
[root@shrishti fat]# modprobe vfat
[root@shrishti fat]#

```

Figure 2: Linux module operations

The `vfat` module depends on the `fat` module, so `fat.ko` needs to be loaded first. To automatically perform decompression and dependency loading, use `modprobe` instead. Note that you shouldn't specify the `.ko` extension to the module's name, when using the `modprobe` command. `rmmmod` is used to unload the modules.

Our first Linux driver

Before we write our first driver, let's go over some concepts. A driver never runs by itself. It is similar to a library that is loaded for its functions to be invoked by a running application. It is written in C, but lacks a `main()` function. Moreover, it will be loaded/linked with the kernel, so it needs to be compiled in a similar way to the kernel, and the header files you can use are only those from the kernel sources, not from the standard `/usr/include`.

One interesting fact about the kernel is that it is an object-oriented implementation in C, as we will observe even with our first driver. Any Linux driver has a constructor and a destructor. The module's constructor is called when the module is successfully loaded into the kernel, and the destructor when `rmmmod` succeeds in unloading the module. These two are like normal functions in the driver, except that they are specified as the `init` and `exit` functions, respectively, by the macros `module_init()` and `module_exit()`, which are defined in the kernel header `module.h`.

```

1  /* ofd.c - Our First Driver code */
2  #include <linux/module.h>
3  #include <linux/version.h>
4  #include <linux/kernel.h>
5
6  static int __init ofd_init(void) /* Constructor */
7  {
8      printk(KERN_INFO "Namaskar: ofd registered");
9      return 0;
10 }
11
12 static void __exit ofd_exit(void) /* Destructor */
13 {
14     printk(KERN_INFO "Alvida: ofd unregistered");
15 }
16
17 module_init(ofd_init);
18 module_exit(ofd_exit);
19
20 MODULE_LICENSE("GPL");
21 MODULE_AUTHOR("Anil Kumar Pugalia <email_at_sarika-pugs_dot_com>");

```

```
22 | MODULE_DESCRIPTION("Our First Driver");
```

Given above is the complete code for our first driver; let's call it `ofd.c`. Note that there is no `stdio.h` (a user-space header); instead, we use the analogous `kernel.h` (a kernel space header). `printk()` is the equivalent of `printf()`. Additionally, `version.h` is included for the module version to be compatible with the kernel into which it is going to be loaded. The `MODULE_*` macros populate module-related information, which acts like the module's "signature".

Building our first Linux driver

Once we have the C code, it is time to compile it and create the module file `ofd.ko`. We use the kernel build system to do this. The following `Makefile` invokes the kernel's build system from the kernel source, and the kernel's `Makefile` will, in turn, invoke our first driver's `Makefile` to build our first driver.

To build a Linux driver, you need to have the kernel source (or, at least, the kernel headers) installed on your system. The kernel source is assumed to be installed at `/usr/src/linux`. If it's at any other location on your system, specify the location in the `KERNEL_SOURCE` variable in this `Makefile`.

```
1 | # Makefile - makefile of our first driver
2 |
3 | # if KERNELRELEASE is defined, we've been invoked from the
4 | # kernel build system and can use its language.
5 | ifneq (${KERNELRELEASE},)
6 |     obj-m := ofd.o
7 | # Otherwise we were called directly from the command line.
8 | # Invoke the kernel build system.
9 | else
10 |     KERNEL_SOURCE := /usr/src/linux
11 |     PWD := $(shell pwd)
12 | default:
13 |     ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} modules
14 |
15 | clean:
16 |     ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} clean
17 | endif
```

With the C code (`ofd.c`) and `Makefile` ready, all we need to do is invoke `make` to build our first driver (`ofd.ko`).

```
$ make
```

```
make -C /usr/src/linux SUBDIRS=... modules
make[1]: Entering directory `/usr/src/linux'
  CC [M]  .../ofd.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      .../ofd.mod.o
  LD [M]  .../ofd.ko
make[1]: Leaving directory `/usr/src/linux'
```

Summing up

Once we have the `ofd.ko` file, perform the usual steps as the root user, or with `sudo`.

```
# su
# insmod ofd.ko
# lsmod | head -10
```

`lsmod` should show you the `ofd` driver loaded.

While the students were trying their first module, the bell rang, marking the end of the session. Professor Gopi concluded, "Currently, you may not be able to observe anything other than the `lsmod` listing showing the driver has loaded. Where's the `printk` output gone? Find that out for yourselves, in the lab session, and update me with your findings. Also note that our first driver is a template for any driver you would write in Linux. Writing a specialised driver is just a matter of what gets filled into its constructor and destructor. So, our further learning will be to enhance this driver to achieve specific driver functionalities."

Share this:



Anil Kumar Pugalia

The author is a freelance trainer in Linux internals, Linux device drivers, embedded Linux and related topics. Prior to this, he had worked at Intel and Nvidia. He has been exploring Linux since 1994. A gold medallist from the Indian Institute of Science, Linux and knowledge-sharing are two of his

many passions.



3