

kmu1990 (<http://habrahabr.ru/users/kmu1990/>) 25 May 2014 TRANSLATION  
(<http://habrahabr.ru/company/spbau/blog/218833/>)

# Writing a File System in Linux Kernel

📁 \*nix

- 
- 
- 
- 
- 
- Who This Article is for
- Introduction
  - Environment Setup
  - Environment Check Up
  - Returning to the File System
- Summary
- Links

## Who This Article is for

There are no complex or difficult concepts in this article, all required is a basic knowledge of the command line, the C language, Makefile and a general understanding about file systems.

In this article I am going to describe the components necessary for development inside the Linux kernel, then we'll write the simplest loadable kernel module and, finally, write a framework for the future file system. It's a module that will register quite a useful (for now) file system in the kernel. The ones familiar with development inside Linux kernel may not find anything interesting here.

## Introduction

A file system is one of the central OS subsystems. File systems continue to develop as operating systems evolve. Currently we have an entire heterogenetic zoo of file systems, from the old "classic" UFS ([http://en.wikipedia.org/wiki/Unix\\_File\\_System](http://en.wikipedia.org/wiki/Unix_File_System)), to the new exotic NILFS (<http://en.wikipedia.org/wiki/NILFS>) (though this idea isn't new at all, look at LFS ([http://en.wikipedia.org/wiki/Log-structured\\_File\\_System\\_\(BSD\)](http://en.wikipedia.org/wiki/Log-structured_File_System_(BSD)))) and BTRFS (<http://en.wikipedia.org/wiki/Btrfs>). We aren't going to try to throw down monsters like ext3/4 and BTRFS. Our file system will be of educational nature, and we will familiarize ourselves with the Linux kernel using its help.

## Environment Setup

Before we get into the kernel, let's prepare all the necessary steps for building our OS module. I use Ubuntu, so I'm going to setup within this environment. Fortunately, it's not difficult at all. To begin with, we're going to need a compiler and building facilities:

```
sudo apt-get install gcc build-essential
```

Then we may need the kernel source code. We'll go the easy route and won't bother rebuilding the kernel from the source. We'll just determine kernel headers; this should be enough to write a loadable module. The headers can be determined the following way:

```
sudo apt-get install linux-headers-$(uname -r)
```

And now I'm going to jump onto my soap box. Rummaging in the kernel on a working machine isn't the smartest idea, so I strongly recommend you perform all these actions within a virtual machine. We won't do anything dangerous so the stored data is safe. But if anything goes wrong, we'll probably have to restart the system. Besides, it's more comfortable to debug the kernel modules in a virtual machine (such as QEMU), though this question won't be considered in the article.

## Environment Check Up

In order to check the environment we'll write and start the kernel module, which won't do anything useful (Hello, World!). Let's consider the module code. I named it super.c (super is derived from superblock):

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

static int __init aufs_init(void)
{
    pr_debug("aufs module loaded\n");
    return 0;
}

static void __exit aufs_fini(void)
{
    pr_debug("aufs module unloaded\n");
}

module_init(aufs_init);
module_exit(aufs_fini);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("kmu");
```

At the very beginning you can see two headers. They are an important part of any loadable module. Then two functions: `aufs_init` and `aufs_fini` follow. They will be called before and after the module roll-out. Some of you may be confused by `__init` label. `__init` is a hint to the kernel that the function is used during module initialization only. It means that after the module initialization it can be unloaded from memory. There is an analogous marker for the data; the kernel can ignore these hints. The reference to `__init` functions and data from the main module code is a potential error. That's why during the module setup it should be checked that there are no such references. If such message is found, the kernel setup system will give out an alert. The similar check is carried out for `__exit` functions and data. If you want to know details about `__init` and `__exit`, you can refer to the source code (<http://lxr.free-electrons.com/source/include/linux/init.h?v=3.14>).

Please note that `aufs_init` returns `int`. Thus the kernel finds out that during the module initialization something went wrong. If the module hasn't returned a zero value, it means that an error occurred during initialization. In order to find out which functions should be called at module loading and unloading, two macros `module_init` and `module_exit` are used. In order to learn details, refer to `lxr`, it's really useful if you want to study the kernel. `pr_debug` – is a function (it's a macro actually, but it doesn't matter for now) of kernel output to the log, it's very similar to the family of `printf` functions with some extensions (for example, for IP and MAC addresses printing. You will find a complete list of modifiers in the documentation of the kernel. Together with `pr_debug`, there is an entire

family of macros: `pr_info`, `pr_warn`, `pr_err` and others. If you are familiar with Linux module development you know about `printk` function. `pr_*` open into `printk` calls, so you can use `printk` instead of them.

## Supercharged Developing

Experience Peak Performance from 100+ Excel  
Engine Controls and Powerful Charts Infragistics  
Ultimate UI

Then there are macros with information about descendants – a license and an author. There are also other macros that allow to store manifold information about the module. For example `MODULE_VERSION`, `MODULE_INFO`, `MODULE_SUPPORTED_DEVICE` and others. By the way, if you're using different form GPL license, you won't be able to use some functions available for GPL modules.

Now let's build and start our module. We'll write Makefile for this. It will build our module:

```
obj-m := aufs.o
aufs-objs := super.o

CFLAGS_super.o := -DDEBUG

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Makefile calls Makefile for building, it should be located in `/lib/modules/$(shell uname -r)/build` catalogue (`uname -r` is a command that returns the version of the started kernel). If the headers (or source codes) of the kernel are located in another catalogue, you should fix it.

`obj-m` allows to state the name of the future module. In our case it will be named `aufs.ko` (ko exactly – from kernel object). `aufs-objs` allows to indicate what source code aufs module should be built from. In our case `super.c` file will be used. You can also indicate different compiler flags, which will be used (in addition to those used by Makefile kernel) for object files building. In our case I pass `-DDEBUG` flag during `super.c` build. If we don't pass the flag, we won't see `pr_debug` output in the system log.

# Voted #1 Helpdesk Software

Signup for the award winning helpdesk, Freshdesk™.  
Trusted by major businesses. Freshdesk

In order to build the module make command should be executed. If everything's fine, aufs.ko file will appear in the catalogue. It's quite easy to load the module: `sudo insmod ./aufs.ko`

In order to make sure that the module is loaded you can look at lsmod command output. `lsmod | grep aufs`

In order to see the system log, dmesg command should be called. We'll see messages from our module in it. It's not difficult to unload the module: `sudo rmmod aufs`

## Returning to the File System

Thus, the environment is adjusted and operates. We know how to build the simplest module, load and unload it. Now we should consider the file system. The file system design should begin "on paper", from a thorough consideration of the data structures being used. But we'll follow a simple path and defer the details of files and folders storage at the disk for the next time. Now we'll write a framework of our future file system.

The life of a file system begins with check-in. You can check-in the file system by calling `register_filesystem` (<http://lxr.free-electrons.com/source/fs/filesystems.c?v=3.14#L69>). We'll register the file system in the function of module initialization. There's `unregister_filesystem` (<http://lxr.free-electrons.com/source/fs/filesystems.c?v=3.14#L101>) for the file system unregistration and we'll call it in `aufs_fini` function of our module.



## Gartner® SIEM Report 2018

Learn why Gartner® named RSA® a  
MQ leader among SIEM vendors.  
[Download the free report.](#)

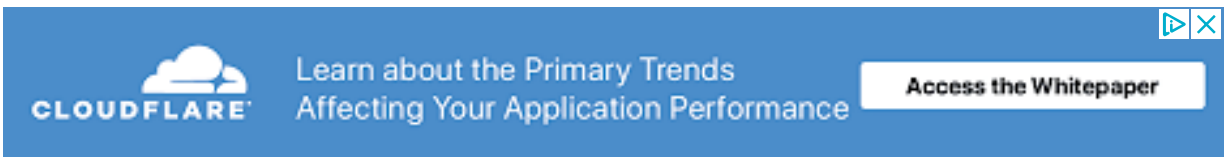
Both functions accept a pointer to `file_system_type` structure as a parameter. It'll "describe" the file system. Consider it as a class of the file system. There're enough fields in the structure, but we're interested in some of them only:

```
static struct file_system_type aufs_type = {  
    .owner = THIS_MODULE,  
    .name = "aufs",  
    .mount = aufs_mount,  
    .kill_sb = kill_block_super,  
    .fs_flags = FS_REQUIRES_DEV,  
};
```

First of all, we are interested in **name** field. It stores the file system name. This name will be used during the assembling.

mount & kill\_sb — are two fields containing pointers to functions. The first function will be called during the file system assembling, the second one during disassembly. It's enough to implement one of them, and we'll use kill\_block\_super instead of the second one, which is provided by the kernel.

fs\_flags — stores different flags. In our case it stores FS\_REQUIRES\_DEV flag, which says that our file system needs a disk for operation (though it's not the case for the moment). You don't have to indicate this flag if you don't want to, everything will operate without it. Finally, owner field is necessary in order to setup a counter of links to the module. The link counter is necessary so that the module wouldn't be unloaded too soon. For example, if the file system has been assembled, the module loading can lead to the crash. The link counter won't allow to unload the module till its being used, i.e. until we disassembly the file system.



Now let's consider aufs\_mount function. It should assemble the device and return the structure describing a root catalogue of the file system. It sounds quite complicated, but, fortunately, the kernel will do it for us as well:

```
static struct dentry *aufs_mount(struct file_system_type *type, int flags,
                                char const *dev, void *data)
{
    struct dentry *const entry = mount_bdev(type, flags, dev,
                                             data, aufs_fill_sb);

    if (IS_ERR(entry))
        pr_err("aufs mounting failed\n");
    else
        pr_debug("aufs mounted\n");
    return entry;
}
```

The biggest part of work happens inside `mount_bdev` function. We are interested in its `aufs_fill_sb` parameter. It's a pointer to the function (again), which will be called from `mount_bdev` in order to initialize the superblock. But before we move on to it, an important file subsystem of the kernel — `dentry` structure — should be considered. This structure represents a section of the path to the file name. For example, if we refer to `/usr/bin/vim` file, we'll have different structure exemplars representing sections of `/` (root catalogue), `bin/` and `vim` path. The kernel supports these structures cache. It allows to quickly find inode (another center structure) by the file (path) name. `aufs_mount` function should return `dentry`, which represents the root catalogue of our file system. `aufs_fill_sb` function will create it.

Thus, for the moment `aufs_fill_sb` is the most important function in our module and it looks like the following:

```
static int aufs_fill_sb(struct super_block *sb, void *data, int silent)
{
    struct inode *root = NULL;

    sb->s_magic = AUFS_MAGIC_NUMBER;
    sb->s_op = &aufs_super_ops;

    root = new_inode(sb);
    if (!root)
    {
        pr_err("inode allocation failed\n");
        return -ENOMEM;
    }

    root->i_ino = 0;
    root->i_sb = sb;
    root->i_atime = root->i_mtime = root->i_ctime = CURRENT_TIME;
    inode_init_owner(root, NULL, S_IFDIR);

    sb->s_root = d_make_root(root);
    if (!sb->s_root)
    {
        pr_err("root creation failed\n");
        return -ENOMEM;
    }

    return 0;
}
```

First of all, we fill `super_block` structure. What kind of structure is it? Usually file systems store in a special place of a disk partition (this place is chosen by file system) the set of file system parameters, such as the block size, the number of occupied/free blocks, file system version, “pointer” to the root catalogue, magic number, by which a driver can check that the necessary file system is stored on the disk. This structure is named `superblock` (look at the picture below). `super_block` structure in Linux kernel is mainly intended for similar goals, we save a magic number in it and dentry for the root catalogue (the one returned by `mount_bdev`).



Besides, in `s_op` field of `super_block` structure we store a pointer to `super_operations` structure — these are `super_block` “class methods”, it's another structure storing a lot of pointers to the functions. I'll make another note here. Linux kernel is written on C, without support of different OOP features from the language side. But we can structure a program following OOP ideas without support from the language. So the structures storing a lot of pointers to the functions can be met quite often in the kernel. It's a method of virtual functions implementation by current facilities.

## Free all-in-one security app

The only cyber security app that's private by design:  
we don't sell your data Avira

Let's get back to `super_block` structure and its “methods”. We're interested in its `put_super` field. We'll save a “destructor” of our superblock in it”

```
static void aufs_put_super(struct super_block *sb)
{
    pr_debug("aufs super block destroyed\n");
}

static struct super_operations const aufs_super_ops = {
    .put_super = aufs_put_super,
};
```

While `aufs_put_super` function does nothing useful, we use it just in order to type in the system log one more line. `aufs_put_super` function will be called within `kill_block_super` (see above) before `super_block` structure deleting. i.e. when disassembling the file system.

Now let's return to the most important `aufs_fill_sb` function. Before we create dentry for the root catalogue we should create an index node (inode) of the root catalogue. Inode structure is probably the most important one in the file system. Each file system object (a file, a folder, a special file, a magazine, etc.) identifies with inode. As well as `super_block`, inode structure reflects the way file systems are stored on a disk. Inode name comes from index node, meaning that it indexes files and folders on a disk. Usually inside inode on a disk a directive to the place file data

are stored on a disk (in which blocks the file content is stored), different access flags (available for read/write/execute), information about the file owner, time of creation/modification/writing/execution and other similar things are stored.

We can't read from the disk yet, so we'll fill inode with fictive data. As time of creation/modification/writing/execution we use the current time and the kernel will assign the owner and access permissions (call `inode_init_owner` function). Finally, create dentry bound with the root inode.

## Skeleton Check Up

The skeleton of our file system is ready. It's time to check it. Building and loading of the file system driver doesn't differ from building and loading of a general module. We'll use loop device instead of a real disk for experiments. It's a "disk" driver, which writes data not on the physical device, but to the file (disk image). Let's create a disk image. It doesn't store any data yet, so it's simple: `touch image`

We should also create a catalogue, which will be an assembling point (root) of our file system:

```
mkdir dir
```

# Free all-in-one security app

The only cyber security app that's private by design:  
we don't sell your data Avira

Now using this image we'll assemble our file system: `sudo mount -o loop -t aufs ./image ./dir`

If the operation ended successfully, we'll see messages from our module in the system log. In order to disassemble the file system we should: `sudo umount ./dir`

Check the system log again.

## Summary

We familiarized ourselves with creation of loadable kernel modules and main structures of the file subsystem. We also wrote a real file system, which can assemble and disassemble only (it's quite silly for the time being, but we're going to fix it in future).

Then we're going to consider data reading from the disk. To begin with we'll define the way data will be stored on disks. We'll also learn how to read superblock and inodes from the disk.

## Links


- The code to the article is available at github ([https://github.com/krinkinmu/aufs\\_les1](https://github.com/krinkinmu/aufs_les1))
- An Indian has quite recently written a simple file system, he has performed a big job (<https://github.com/psankar/simplefs>)
- I understand that it's not very correct educationally to send the newcomers to the kernel source code (though it's useful to read them); I still recommend all the interested ones to look at the source code of a very simple ramfs (<http://lxr.free-electrons.com/source/fs/ramfs/>) file system. Besides, unlike our file system, ramfs doesn't use a disk. It stores everything in the memory.


 Share ([https://www.facebook.com/sharer/sharer.php?](https://www.facebook.com/sharer/sharer.php?u=https%3A%2F%2Fkukuruku.co%2F&t=)

[u=https%3A%2F%2Fkukuruku.co%2F&t=](https://www.facebook.com/sharer/sharer.php?u=https%3A%2F%2Fkukuruku.co%2F&t=))

 Tweet ([https://twitter.com/intent/tweet?](https://twitter.com/intent/tweet?source=https%3A%2F%2Fkukuruku.co%2F&text=%20https%3A%2F%2Fkukuruku.co%2F&via=kukuruku)

[source=https%3A%2F%2Fkukuruku.co%2F&text=%20https%3A%2F%2Fkukuruku.co%2F&via=kukuruku](https://twitter.com/intent/tweet?source=https%3A%2F%2Fkukuruku.co%2F&text=%20https%3A%2F%2Fkukuruku.co%2F&via=kukuruku)

 +1 (<https://plus.google.com/share?url=https%3A%2F%2Fkukuruku.co%2F>)

 Reddit (<http://www.reddit.com/submit?url=https%3A%2F%2Fkukuruku.co%2F&title=>)

 Share ([http://www.linkedin.com/shareArticle?](http://www.linkedin.com/shareArticle?mini=true&url=https%3A%2F%2Fkukuruku.co%2F&title=&summary=&source=https%3A%2F%2Fkukuruku)

[mini=true&url=https%3A%2F%2Fkukuruku.co%2F&title=&summary=&source=https%3A%2F%2Fkukuruku](http://www.linkedin.com/shareArticle?mini=true&url=https%3A%2F%2Fkukuruku.co%2F&title=&summary=&source=https%3A%2F%2Fkukuruku)




(<http://habrahabr.ru/users/kmu1990/>) kmu1990 (<http://habrahabr.ru/users/kmu1990/>)

## Comments



Github flavored markdown is allowed.

Post Comment

Flux 2 January 2017  3,751

## Ropes — Fast Strings

(<https://kukuruku.co/post/ropes-fast-strings/>)