Creating Linux virtual filesystems

[Posted November 11, 2003 by corbet]

This article is part of the LWN Porting Drivers to 2.6 series.

[This article has been reworked to reflect changes in the libfs interface; those who are interested can still read the original version.]

Linus and numerous other kernel developers dislike the `ioctl()` system call, seeing it as an uncontrolled way of adding new system calls to the kernel. Putting new files into `/proc` is also discouraged, since that area is seen as being a bit of a mess. Developers who populate their code with `ioctl()` implementations or `/proc` files are often encouraged to create a standalone virtual filesystem instead. Filesystems make the interface explicit and visible in user space; they also make it easier to write scripts which perform administrative functions. But the writing of a Linux filesystem can be an intimidating task. A developer who has spent some time just getting up to speed on the driver interface can be forgiven for balking at having to learn the VFS API as well.

The 2.6 kernel contains a set of routines called "libfs" which is designed to make the task of writing virtual filesystems easier. libfs handles many of the mundane tasks of implementing the Linux filesystem API, allowing non-filesystem developers to concentrate (mostly) on the specific functionality they want to provide. What it lacks, however, is documentation. This article is an attempt to fill in that gap a little bit.

The task we will undertake is not particularly ambitious: export a simple filesystem (of type "lwnfs") full of counter files. Reading one of these files yields the current value of the counter, which is then incremented. This leads to the following sort of exciting interaction:

```
    # cat /lwnfs/counter
    0
    # cat /lwnfs/counter
    1
    # ...
```

Your author was able to amuse himself well into the thousands this way; some users may tire of this game sooner, however. The impatient can get to higher values more quickly by writing to the counter file:

```
    # echo 1000 > /lwnfs/counter
    # cat /lwnfs/counter
    1000
    #
```

OK, so the Linux distributors will probably not get to excited about advertising the new "lwnfs" capability. But it works as a way of showing how to create virtual filesystems. For those who are interested, the full source is available.

Initialization and superblock setup

So let's get started. A loadable module which implements a filesystem must, at load time, register that filesystem with the VFS layer. The lwnfs module initialization code is simple:

```
    static int __init lfs_init(void)
    {
            return register_filesystem(&lfs_type);
    }
    module_init(lfs_init);
```

The `lfs_type` argument is a structure which is set up as follows:

```
    static struct file_system_type lfs_type = {
            .owner      = THIS_MODULE,
            .name       = "lwnfs",
            .get_sb     = lfs_get_super,
            .kill_sb    = kill_litter_super,
    };
```

This is the basic data structure which describes a filesystem type to the kernel; it is declared in `<linux/fs.h>`. The `owner` field is used to manage the module's reference count, preventing unloading of the module while the filesystem code is in use. The `name` is what eventually ends up on a `mount` command line in user space. Then there are two functions for managing the filesystem's superblock - the root of the filesystem data structure. `kill_litter_super()` is a generic function provided by the VFS; it simply cleans up all of the in-core structures when the filesystem is unmounted; authors of simple virtual filesystems need not worry about this aspect of things. (It *is* necessary to unregister the filesystem at unload time, of course; see the source for the lwnfs exit function).

In many cases, the creation of the superblock must be done by the filesystem programmer -- but see the "a simpler way" section below. This task involves a bit of boilerplate code. In this case, `lfs_get_super()` hands off the task as follows:

```
static struct super_block *lfs_get_super(struct file_system_type *fst,
               int flags, const char *devname, void *data)
{
        return get_sb_single(fst, flags, data, lfs_fill_super);
}
```

Once again, `get_sb_single()` is generic code which handles much of the superblock creation task. But it will call `lfs_fill_super()`, which performs setup specific to our particular little filesystem. It's prototype is:

```
    static int lfs_fill_super (struct super_block *sb,
                               void *data, int silent);
```

The in-construction superblock is passed in, along with a couple of other arguments that we can ignore. We do have to fill in some of the superblock fields, though. The code starts out like this:

```
        sb->s_blocksize = PAGE_CACHE_SIZE;
        sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
        sb->s_magic = LFS_MAGIC;
        sb->s_op = &lfs_s_ops;
```

Most virtual filesystem implementations have something that looks like this; it's just setting up the block size of the filesystem, a "magic number" to recognize superblocks by, and the superblock operations. These operations need not be written for a simple virtual filesystem - libfs has the stuff that is needed. So `lfs_s_ops` is defined (at the top file level) as:

```
    static struct super_operations lfs_s_ops = {
            .statfs         = simple_statfs,
            .drop_inode     = generic_delete_inode,
    };
```

Creating the root directory

Getting back into `lfs_fill_super()`, our big remaining task is to create and populate the root directory for our new filesystem. The first step is to create the inode for the directory:

```
        root = lfs_make_inode(sb, S_IFDIR | 0755);
        if (! root)
                goto out;
        root->i_op = &simple_dir_inode_operations;
        root->i_fop = &simple_dir_operations;
```

`lfs_make_inode()` is a boilerplate function that we will look at eventually; for now, just assume that it returns a new, initialized inode that we can use. It needs the superblock and a `mode` argument, which is just like the mode value returned by the `stat()` system call. Since we passed `S_IFDIR`, the returned inode will describe a directory. The file and directory operations that we assign to this inode are, again, taken from libfs.

This directory inode must be put into the directory cache (by way of a "dentry" structure) so that the VFS can find it; that is done as follows:

```
        root_dentry = d_alloc_root(root);
        if (! root_dentry)
                goto out_iput;
        sb->s_root = root_dentry;
```

Creating files

The superblock now has a fully initialized root directory. All of the actual directory operations will be handled by libfs and the VFS layer, so life is easy. What libfs cannot do, however, is actually put anything of interest into that root directory – that's our job. So the final thing that `lfs_fill_super()` does before returning is to call:

```
        lfs_create_files(sb, root_dentry);
```

In our sample module, `lfs_create_files()` creates one counter file in the root directory of the filesystem, and another in a subdirectory. We'll look mostly at the root-level file. The counters are implemented as `atomic_t` variables; our top-level counter (called, with great imagination, "`counter`") is set up as follows:

```
    static atomic_t counter;

    static void lfs_create_files (struct super_block *sb,
                                  struct dentry *root)
    {
            /* ... */
            atomic_set(&counter, 0);
            lfs_create_file(sb, root, "counter", &counter);
            /* ... */
    }
```

`lfs_create_file` does the real work of making a file in a directory. It has been made about as simple as possible, but there are still a few steps to be performed. The function starts out as:

```
static struct dentry *lfs_create_file (struct super_block *sb,
                struct dentry *dir, const char *name,
                atomic_t *counter)
{
        struct dentry *dentry;
        struct inode *inode;
        struct qstr qname;
```

Arguments include the usual superblock structure, and `dir`, the dentry for the directory that will contain this file. In this case, `dir` will be the root directory we created before, but it could be any directory within the filesystem.

Our first task is to create a directory entry for the new file:

```
        qname.name = name;
        qname.len = strlen (name);
        qname.hash = full_name_hash(name, qname.len);
        dentry = d_alloc(dir, &qname);
```

The setting up of `qname` just hashes the file name so that it can be found quickly in the dentry cache. Once that's done, we create the entry within our parent `dir`. The file also needs an inode, which we create as follows:

```
        inode = lfs_make_inode(sb, S_IFREG | 0644);
        if (! inode)
                goto out_dput;
        inode->i_fop = &lfs_file_ops;
        inode->u.generic_ip = counter;
```

Once again, we call `lfs_make_inode` (which we will look at shortly, honest), but this time we use it to create a regular file. The key to the creation of special-purpose files in virtual filesystems is to be found in the other two assignments:

- The `i_fop` field is set up with our file operations which will actually implement reads and writes on the counter.
- We use the `u.generic_ip` pointer in the inode to stash aside a pointer to the `atomic_t` counter associated with this file.

In other words, `i_fop` defines the behavior of this particular file, and `u.generic_ip` is the file-specific data. All virtual filesystems of interest will make use of these two fields to set up the required behavior.

The last step in creating a file is to add it to the dentry cache:

```
        d_add(dentry, inode);
        return dentry;
```

Putting the inode into the dentry cache allows the VFS to find the file without having to consult our filesystem's directory operations. And that, in turn, means our filesystem does not need to *have* any directory operations of interest. The entire structure of our virtual filesystem lives in the kernel's cache structure, so our module need not remember the structure of the filesystem it has set up, and it need not implement a lookup operation. Needless to say, that makes life easier.

Inode creation

Before we get into the actual implementation of the counters, it's time to look at `lfs_make_inode()`. The function is pure boilerplate; it looks like:

```
static struct inode *lfs_make_inode(struct super_block *sb, int mode)
{
        struct inode *ret = new_inode(sb);

        if (ret) {
                ret->i_mode = mode;
                ret->i_uid = ret->i_gid = 0;
                ret->i_blksize = PAGE_CACHE_SIZE;
                ret->i_blocks = 0;
                ret->i_atime = ret->i_mtime = ret->i_ctime = CURRENT_TIME;
        }
        return ret;
}
```

It simply allocates a new inode structure, and fills it in with values that make sense for a virtual file. The assignment of `mode` is of interest; the resulting inode will be a regular file or a directory (or something else) depending on how `mode` was passed in.

Implementing file operations

Up to this point, we have seen very little that actually makes the counter files work; it's all been VFS boilerplate so that we have a little filesystem to put those counters into. Now the time has come to see how the real work gets done.

The operations on the counters themselves are to be found in the `file_operations` structure that we associate with the counter file inodes:

```
    static struct file_operations lfs_file_ops = {
            .open       = lfs_open,
            .read       = lfs_read_file,
            .write      = lfs_write_file,
    };
```

A pointer to this structure, remember, was stored in the inode by `lfs_create_file()`.

The simplest operation is `open()`:

```
    static int lfs_open(struct inode *inode, struct file *filp)
    {
            filp->private_data = inode->u.generic_ip;
            return 0;
    }
```

The only thing this function need do is copy the pointer to the `atomic_t` pointer over into the `file` structure, which makes it a bit easier to get at.

The interesting work is done by the `read()` function, which must increment the counter and return its value to the user space program. It has the usual `read()` operation prototype:

```
    static ssize_t lfs_read_file(struct file *filp, char *buf,
                                 size_t count, loff_t *offset)
```

It starts by reading and incrementing the counter:

```
        atomic_t *counter = (atomic_t *) filp->private_data;
        int v = atomic_read(counter);
        atomic_inc(counter);
```

This code has been simplified a bit; see the module source for a couple of grungy, irrelevant details. Some readers will also notice a race condition here: two processes could read the counter before either increments it; the result would be the same counter value returned twice, with certain dire results. A serious module would probably serialize access to the counter with a spinlock. But this is supposed to be a simple demonstration.

So anyway, once we have the value of the counter, we have to return it to user space. That means encoding it into character form, and figuring out where and how it fits into the user-space buffer. After all, a user-space program can seek around in our virtual file.

```
        len = snprintf(tmp, TMPSIZE, "%d\n", v);
        if (*offset > len)
                return 0;
        if (count > len - *offset)
                count = len - *offset;
```

Once we've figured out how much data we can copy back, we just do it, adjust the file offset, and we're done.

```
        if (copy_to_user(buf, tmp + *offset, count))
                return -EFAULT;
        *offset += count;
        return count;
```

Then, there is `lfs_write_file()`, which allows a user to set the value of one of our counters:

```
static ssize_t lfs_write_file(struct file *filp, const char *buf,
                size_t count, loff_t *offset)
{
        atomic_t *counter = (atomic_t *) filp->private_data;
        char tmp[TMPSIZE];

        if (*offset != 0)
                return -EINVAL;
        if (count >= TMPSIZE)
                return -EINVAL;

        memset(tmp, 0, TMPSIZE);
        if (copy_from_user(tmp, buf, count))
                return -EFAULT;
        atomic_set(counter, simple_strtol(tmp, NULL, 10));
        return count;
}
```

That is just about it. The module also defines `lfs_create_dir`, which creates a directory in the filesystem; see the full source for how that works.

A simpler way

The above example contains a great deal of scary-looking boilerplate code. That boilerplate will be necessary for many applications, but there is a shortcut that will work for many others. If you know at compile time which files you wish to create, and you do not need to make subdirectories, read on for the easier way.

In this section, we'll talk about a different version of the lwnfs module - one which eliminates about 1/3 of the code. It implements a simple array of four counters, with no subdirectories. Once again, full source is available if you are interested.

Above, we looked at a function called `lfs_fill_super()`, which fills in the filesystem superblock, creates the root directory, and populates it with files. In the simpler version, the entire function becomes the following:

```
    static int lfs_fill_super(struct super_block *sb, void *data, int silent)
    {
            return simple_fill_super(sb, LFS_MAGIC, OurFiles);
    }
```

`simple_fill_super()` is a libfs function which does almost everything we need. Its actual prototype is:

```
    int simple_fill_super(struct super_block *sb, int magic,
                          struct tree_descr *files);
```

The `struct super_block` argument can be passed directly through, and `magic` is the same magic number we saw above. The `files` argument describes which files should be created in the filesystem; the relevant structure is defined as follows:

```
struct tree_descr {
        char *name;
        struct file_operations *ops;
        int mode;
};
```

The arguments should be fairly obvious by now; each structure gives the name of the file to be created, the file operations to associate with the file, and the protection bits for the file. There are, however, a couple of quirks about how the array of `tree_descr` structures should be built:

- Entries which are filled with `NULL`s (more strictly, where `name` is `NULL`) are simply ignored. Do not try to end the list with a `NULL-filled` structure, unless you like decoding oops listings.
- The list is terminated, instead, by an entry that sets `name` to the empty string.
- The entries correspond directly to the inode numbers which will be assigned to the resulting files. This knowledge can be used to figure out, in the file operations code, which file is being opened. But this feature also implies that the first entry in the list cannot be used, since the filesystem root directory will take inode zero. So, when you create your `tree_descr` list, the first entry should be `NULL`.

Having painfully learned all of the above, your author has set up the list for the four "counter" files as follows:

```
static struct tree_descr OurFiles[] = {
        { NULL, NULL, 0 },                 /* Skipped */
        { .name = "counter0",              /* Inode 1 */
          .ops = &lfs_file_ops,
          .mode = S_IWUSR|S_IRUGO },
        { .name = "counter1",              /* Inode 2 */
          .ops = &lfs_file_ops,
          .mode = S_IWUSR|S_IRUGO },
        { .name = "counter2",              /* Inode 3 */
          .ops = &lfs_file_ops,
          .mode = S_IWUSR|S_IRUGO },
        { .name = "counter3",              /* Inode 4 */
          .ops = &lfs_file_ops,
          .mode = S_IWUSR|S_IRUGO },
        { "", NULL, 0 }            /* Terminates the list */
};
```

Once the call to `simple_fill_super()` returns, the work is done and your filesystem is live. The only remaining detail might be in your `open()` method; if you have multiple files sharing the same `file_operations` structure, you will need to figure out which one is actually being acted upon. The key here is the inode number, which can be found in the `i_ino` field. The modified version of `lfs_open()` finds the right counter as follows:

```
static int lfs_open(struct inode *inode, struct file *filp)
{
        if (inode->i_ino > NCOUNTERS)
                return -ENODEV;  /* Should never happen.  */
        filp->private_data = counters + inode->i_ino - 1;
        return 0;
}
```

The `read()` and `write()` functions use the `private_data` field, and thus need not be modified from the previous version.

Conclusion

The libfs code, as demonstrated here, is sufficient for a wide variety of driver-specific virtual filesystems. Further examples can be found in the 2.5 kernel source in a few places:

- `drivers/hotplug/pci_hotplug_core.c`
- `drivers/usb/core/inode.c`
- `drivers/oprofile/oprofilefs.c`
- `fs/ramfs/inode.c`
- `fs/nfsd/nfsctl.c` (`simple_fill_super()` example)

...and in a few other spots – `grep` is your friend.

Keep in mind that the 2.6 driver model code makes it easy for drivers to export information within its own virtual filesystem; for many applications, that will be the preferred way of making information available to user space. The Driver Porting Series has several articles on the driver model and sysfs. For cases where only a custom filesystem will do, however, libfs makes the task (relatively) easy.