Putting a filesystem into a device driver

Greg Kroah-Hartman *
IBM Corp.
greg@kroah.com or gregkh@us.ibm.com

Abstract

For a while there was a freeze on assigning new major and minor numbers in the kernel, so developers had to use a different way to have a device driver interact with userspace. One of the ways this can be done is by embedding a filesystem into the device driver. This paper will show how this can be easily done for both the 2.4 and 2.5 kernel trees.

It will cover what the basic requirements for a filesystem are, how to create the internal kernel structures and register them properly, and how to create a filesystem that is contained within a removable module. Where possible it will highlight the differences between the 2.4 and 2.5 kernel versions of the VFS layer, and how it can help do the main work for the driver. For drivers that only need to be in 2.5 and beyond, the `sysfs` filesystem will be discussed as an alternative to creating a separate filesystem.

1 Introduction

On May 14, 2001, H. Peter Anvin announced to the linux-kernel mailing list:

> Linus Torvalds has requested a moratorium on new device number assignments. His hope is that a new and better method for device space handing will emerge as a result.

Peter is the "Linux Assigned Names and Numbers Authority". This means that all kernel driver authors had to go through him to get a major and minor number pair for their drivers. With this announcement, a freeze was made on assigning new numbers. Naturally this caused a lot of discussion on what this "better method" for device space handling would be. One viewpoint that emerged from the discussion was the fact that a driver could implement a filesystem to control the user space interaction with the driver.

2 `pcihpfs`

During this time, the PCI Hotplug driver that was written by Compaq for their servers was being cleaned up for submission to the main kernel tree. A PCI Hotplug driver allows the user to shutdown a PCI card while the machine is running, pull it out, replace it with another one, and then power that card back on, if the proper PCI controller hardware is present. This is very useful for servers that can not be shut down, but need to have new network cards added, faulty devices removed, and other service type operations.

The PCI Hotplug driver was originally written to interact with user space through a single character device node. `ioctl(2)` calls were made to the device node to shutdown PCI slots, power up PCI slots, turn PCI slot indicator lights on and off, and to run different manufacturing tests on the device. To get information about the number of different PCI slots in the system, and the state of the slots (power and indicator status), a `/proc` directory tree was used. This directory tree was read only.

As work progressed splitting the PCI Hotplug core functionality out of the Compaq driver, so that other PCI Hotplug drivers would have a common interface to the user, it was determined that a single filesystem would be a better fit to both show PCI slot information, and to allow user control of the slots. All information and control over the driver

*This work represents the view of the author and does not necessarily represent the view of IBM.

would be made from one place, instead of having two different types of interfaces.

The PCI Hotplug driver core has been merged into the main kernel tree as of 2.4.15, and it exports a filesystem called `pcihpfs` that is used to control the driver. When the filesystem is mounted, traditionally at `/proc/bus/pci/slots`, a tree is created that looks something like the following:

```
.
|-- slot3
|   |-- adapter
|   |-- attention
|   |-- latch
|   |-- power
|   `-- test
|-- slot4
|   |-- adapter
|   |-- attention
|   |-- latch
|   |-- power
|   `-- test
|-- slot5
|   |-- adapter
|   |-- attention
|   |-- latch
|   |-- power
|   `-- test
`-- slot6
    |-- adapter
    |-- attention
    |-- latch
    |-- power
    `-- test
```

The directories called 3, 4, 5, and so on, are the physical numbers of the PCI slots. Every file in a slot directory can be read to get the value for that bit of information about the slot. The files `power` and `attention` can be written to set the power (0 or 1) or attention (0 or 1) values. The `test` file is used to send hardware test commands to the hardware. The `adapter` file describes if an adapter is present in that slot or not, and the "latch" file describes the position of the physical latch (if any) for that slot.

So the power in slot 5 can be enabled by running:

```
echo 1 > 5/power
```

from the `pcihpfs` root. If a PCI card is present in that slot, the whole PCI initialization sequence will happen for that card, including calling out to `/sbin/hotplug` with the PCI info so that the module for that device can be automatically loaded by the system.

Because of this filesystem, a user space program does not have to make special `ioctl()` calls to a character device, enabling users to have a wider range of options for how they want to control their devices.

3 Creating a filesystem

A filesystem must be declared within the driver. To do this, create a `struct file_system_type` variable, and fill in some of the fields. An example of this can be found in the `drivers/hotplug/pci_hotplug_core.c` file:

```
static struct file_system_type pcihpfs_type = {
        .owner =        THIS_MODULE,
        .name =         "pcihpfs",
        .get_sb =       pcihpfs_get_sb,
        .kill_sb =      kill_litter_super,
};
```

The name field is set to `pcihpfs` which will be used by users in mounting the filesystem (so choose a name that makes sense, and is not currently in use by any other filesystem in the kernel.) By setting the `kill_sb` function pointer to `kill_litter_super` the driver specifies that it wants the filesystem to keep the tree in the dcache. This is because the filesystem will live completely in ram, and will not have a backing store of the data on any physical device (like a disk).

The `get_sb` field of the `pcihpfs_fs_type` points to the function that will be called when the kernel wants to read the superblock of the filesystem. A superblock is the structure in a filesystem that is used to describe the entire filesystem. The kernel will call this function when the filesystem is asked to be mounted. When this function is called, the kernel needs to be told exactly what the filesystem looks like.

But before the filesystem can be mounted, the driver needs to tell the kernel that the filesystem is present. This is done with a simple call to `register_filesystem()` with the `file_system_type` as the only parameter. This is done in the pci_hotplug module's initialization function with the following bit of code:

```
result = register_filesystem(&pcihpfs_fs_type);
if (result) {
        err("register_filesystem failed with"
            " %d\n", result);
        goto exit;
}
```

Likewise, when the pci_hotplug module is being shutdown, the filesystem type is unregistered with the following single line of code:

```
unregister_filesystem(&pcihpfs_fs_type);
```

For the 2.4 kernel, declaring a filesystem is done a bit differently. The DECLARE_FSTYPE macro is used instead of creating the struct file_system_type variable directly:

```
static DECLARE_FSTYPE(pcihpfs_fs_type,
                      "pcihpfs",
                      pcihpfs_read_super,
                      FS_SINGLE | FS_LITTER);
```

The FS_SINGLE flag means that for this filesystem, there will only be one instance of the superblock. This means that wherever the filesystem is mounted in the system, all mount points will point to the same location in the filesystem (the same filesystem can be mounted at different points in the directory tree at the same time.) If this is not specified, every time the filesystem is mounted, a new superblock is created, requiring all of the virtual files to be recreated. Most drivers that implement filesystems want the single instance of the superblock. For 2.5 this flag is not specified in the struct file_system_type variable, but it is specified within the function that is called to get the superblock of the filesystem.

The FS_LITTER option means the same as the kill_litter_super function being set in the 2.5 kernel.

4 Mounting the filesystem

After the filesystem is registered, the driver can create some virtual files that will be used by a user to read and write values to the driver. If a user mounts the filesystem, before the driver creates a file, the kernel will have already created the filesystem at some virtual location. If the filesystem has not been mounted by a user, the driver has to get the kernel to mount the filesystem internally before it can create a file. There are three ways to do this.

The first, and easiest method, is to call kern_mount() right after register_filesystem() is called. This mounts the kernel internally, and allows files to be created and removed from that point on. The main disadvantage of this method is that the module that implemented the filesystem can not be unloaded until the internal mount is unmounted, effectively locking the module and filesystem into memory forever. For filesystems that are not meant to be unloaded, this is acceptable. An example of this is the sysfs filesystem in the 2.5 kernel.

The second method is to wait until the filesystem is really mounted and then create all of the needed files. The get_sb function is called by the kernel when the filesystem is mounted, so at that moment the driver could create the files. (For 2.4, it's when the read_super function is called.) To do this properly, it would require a lot of work to be done at mount time, and to constantly be aware of if the filesystem is currently mounted or not. This can be a big problem if files need to be added or removed at different points in time (like when devices are added or removed from the system.) The usbfs filesystem in the 2.2 and 2.4 kernels is an example of a virtual filesystem that implements operates this way.

But if the filesystem is required to live in a module that can be unloaded from memory, and the number of different files that need to be created or removed is too difficult to keep track of, there is a third method that can be used. This involves telling the kernel to internally mount the filesystem, but still allow it to be unmounted at a later time. The code in Figure **??** shows how to accomplish this.

This code is also a good example of how to do proper locking techniques for when the kernel is running on a multiple processor machine.

First the spin lock, mount_lock is grabbed with the line:

```
spin_lock(\&mount_lock);
```

```
static int get_mount (struct file_system_type *fs_type, struct vfsmount **mount, int *mount_count)
{
	struct vfsmount *mnt;

	spin_lock (&mount_lock);
	if (*mount) {
		mntget(*mount);
		++(*mount_count);
		spin_unlock (&mount_lock);
		goto go_ahead;
	}

	spin_unlock (&mount_lock);
	mnt = kern_mount (fs_type);
	if (IS_ERR(mnt)) {
		err ("could not mount the fs...erroring out!\n");
		return -ENODEV;
	}
	spin_lock (&mount_lock);
	if (!*mount) {
		*mount = mnt;
		++(*mount_count);
		spin_unlock (&mount_lock);
		goto go_ahead;
	}
	mntget(*mount);
	++(*mount_count);
	spin_unlock (&mount_lock);
	mntput(mnt);

go_ahead:
	dbg("mount_count = %d", *mount_count);
	return 0;
}
```

Figure 1: `get_mount` from `drivers/usb/core/inode.c`

This lock is used to protect the internal count of how many times the filesystem has been mounted. Previously it was stated that it would not be necessary to keep track of if the filesystem was mounted or not. This simple function, combined with a simple function to unmount the filesystem described later, is much easier to understand and work with, than the option of trying to determine if it has been mounted by a user or not. See the code in `drivers/usb/inode.c` in the 2.4.18 and earlier kernels for an example of what is needed to do this properly.

After `mount_lock` has been grabbed, the internal mount variable is checked:

```
	if (*mount) {
		mntget(*mount);
		++(*mount_count);
		spin_unlock (&mount_lock);
		goto go_ahead;
	}
```

If it has been set, the code calls `mntget()` to increment the internal mount count variable (`mntget()` is a simple inline function in the `include/linux/mount.h` file). The the internal count variable is incremented, the `mount_lock` is unlocked, and the function exits by jumping to the `go_ahead` label.

If this filesystem has not been mounted, more work needs to occur. The `mount_lock` is unlocked:

```
	spin_unlock (&mount_lock);
```

and `kern_mount` is called to mount the filesystem internally:

```
mnt = kern_mount (fs_type);
if (IS_ERR(mnt)) {
        err ("could not mount the fs..."
             "erroring out!\n");
        return -ENODEV;
}
```

The mount_lock is unlocked because the kern_mount() function can possibly take a long time and even cause the kernel to sleep and schedule another process. A spin lock can not be held if schedule() can be called while the lock is held. If this happens, the kernel could easily deadlock.

After the filesystem has been mounted, mount_lock is grabbed again:

```
spin_lock (&mount_lock);
```

and then the internal mount variable is checked again to determine if it is still zero:

```
if (!*mount) {
        *mount = mnt;
        ++(*mount_count);
        spin_unlock (&mount_lock);
        goto go_ahead;
}
```

Why should the variable be checked again, as it was just checked a few lines ago? This is done because the call to kern_mount() that was previously called could sleep. If that happened, another process could come through this same piece of code and the filesystem could have already been successfully mounted. This is why the variable must be checked again.

If another process has not come through and mounted the filesystem, the pointer to the now mounted filesystem is saved off for other functions to later use, the internal counter is incremented, mount_lock is unlocked, and the function exits.

But if another process has already mounted the filesystem, the code does:

```
mntget(*mount);
++(*mount_count);
spin_unlock (&mount_lock);
mntput(mnt);
```

which matches what was originally done in the same situation, back up at the beginning of the function.

The code to unmount the filesystem is much simpler, and can be seen in Figure ??. In this function, the mount_lock is grabbed (this is the same lock used when mounting the filesystem). Then the count of the number of times the filesystem was mounted is decremented. Because of this, the put_mount needs to be called as many times as get_mount is called. After this, the mount_lock is unlocked, and the kernel is told to unmount the filesystem with a call to mntput().

The code for get_mount and put_mount will work properly for both the 2.4 and 2.5 kernels.

5 Creating the superblock

When the kernel wants to mount the filesystem, virtually due to a call to kern_mount(), or because a user mounted it first, the superblock get function is called by the VFS core. For 2.5 kernels, this is the get_sb callback, and for the 2.4 kernel, it is the function specified in the DECLARE_FSTYPE macro.

A superblock is an object needed by the kernel VFS (virtual file system layer) that describes a mounted filesystem. If a filesystem is based on a disk, this usually corresponds to data stored on the disk in a filesystem control block. For a virtual filesystem, this information must be created based on the information that the driver wants to place into the filesystem.

For the pcihpfs code in the 2.5 kernel, the get_sb callback is a very simple function:

```
static struct super_block *pcihpfs_get_sb (
        struct file_system_type *fs_type,
        int flags, char *dev_name, void *data)
{
        return get_sb_single(fs_type,
                             flags,
                             data,
                             pcihpfs_fill_super);
}
```

The call to get_sb_single() tells the VFS layer that the driver wants a single instance of this filesystem in memory, exactly like the FS_SINGLE flag in

```
static void put_mount (struct vfsmount **mount, int *mount_count)
{
        struct vfsmount *mnt;

        spin_lock (&mount_lock);
        mnt = *mount;
        --(*mount_count);
        if (!(*mount_count))
                *mount = NULL;

        spin_unlock (&mount_lock);
        mntput(mnt);
        dbg("mount_count = %d", *mount_count);
}
```

Figure 2: `put_mount` from `drivers/usb/core/inode.c`

the 2.4 `DEVCLARE_FSTYPE` macro stated.

The `pcihpfs_fill_super` function is where all of the superblock information for the filesystem is created. This includes information describing what the filesystem looks like and where to find the functions that the kernel will later call during the lifetime of the filesystem. This is done with the following lines of code:

```
        sb->s_blocksize = PAGE_CACHE_SIZE;
        sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
        sb->s_magic = PCIHPFS_MAGIC;
        sb->s_op = &pcihpfs_ops;
```

This code specifies that the filesystem's block size is equal to the page cache size. The filesystem's magic number is also set up. This number must be unique across all filesystems in the kernel. The pointer to the list of `struct super_operations` is also specified.

Then the superblock's root inode is initialized:

```
inode = pcihpfs_get_inode(sb,
                          S_IFDIR | 0755,
                          0);

if (!inode) {
        dbg("%s: could not get inode!\n",
            __FUNCTION__);
        return -ENOMEM;
}
```

`pcihpfs_get_inode()` will be described down below. If that function succeeds, the root dentry for the inode that was created is allocated, and that dentry is saved into the superblock structure:

```
root = d_alloc_root(inode);
if (!root) {
        dbg("%s: could not get root dentry!\n",
            __FUNCTION__);
        iput(inode);
        return -ENOMEM;
}
sb->s_root = root;
```

This completes everything that is needed to initialize the superblock structure, and now the kernel has successfully mounted the filesystem.

6 Creating inodes

The kernel VFS works based on a inode structure. An inode stores general information about a specific file. For a disk based filesystem, an inode would usually correspond to a specific file control block that would be stored on a disk. For a virtual filesystem, it refers to a specific file.

Inodes are created by the filesystem when asked to. The `pcihpfs` file system does this in the `pcihpfs_get_inode()` function. filesystem. This function is shown in Figure **??**.

The `pcihpfs_get_inode` function first calls the kernel's `new_inode()` function to have a new inode structure initialized and created. If this succeeds, the function proceeds to fill up a number of the inode structure's fields with needed information. The `i_uid` and `i_gid` members are set to the current process's uid and gid values. This insures that whoever

```
static struct inode *pcihpfs_get_inode (struct super_block *sb, int mode, dev_t dev)
{
        struct inode *inode = new_inode(sb);

        if (inode) {
                inode->i_mode = mode;
                inode->i_uid = current->fsuid;
                inode->i_gid = current->fsgid;
                inode->i_blksize = PAGE_CACHE_SIZE;
                inode->i_blocks = 0;
                inode->i_rdev = NODEV;
                inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
                switch (mode & S_IFMT) {
                default:
                        init_special_inode(inode, mode, dev);
                        break;
                case S_IFREG:
                        inode->i_fop = &default_file_operations;
                        break;
                case S_IFDIR:
                        inode->i_op = &pcihpfs_dir_inode_operations;
                        inode->i_fop = &simple_dir_operations;

                        /* directory inodes start off with i_nlink == 2 (for "." entry) */
                        inode->i_nlink++;
                        break;
                }
        }
        return inode;
}
```

Figure 3: `pcihpfs_get_inode` from `drivers/hotplug/pci_hotplug_core.c`

has the permission to create the inode, can later access it. The `i_atime`, `i_mtime`, and `i_ctime` members refer to the inode's access time, last modified time, and time of last change for the file. These are all set to the current time. If this inode is a "regular" file type, then the set of functions that should be called whenever the inode is acted upon (for open, write, read, etc.) is set to point to the `default_file_operations` structure. If this inode is a directory inode, the file operations is pointed to a default set of directory inode functions. And if the inode is neither a "regular" inode, or a directory inode, then the kernel initializes it with a call to `init_special_inode()`.

7 Creating virtual files

Now that the filesystem is internally mounted, virtual files can be created. To do this, a number of different options for a file must be determined. The `fs_create_file()` function shown in Figure **??** shows all of the different parameters needed.

`fs_create_file` expects the following parameters:

- the name of the file
- the permission (mode) of the file
- the parent directory for the file (if this is NULL, then the file is created in the root directory of the filesystem)
- a pointer to a blob of data that will be assigned to this file
- a pointer to a `struct file_operations` that will be used for this file
- and the user and group ids for the file.

The function takes the name, mode, and parent, and calls the `fs_create_by_name()` function

```
static struct dentry *fs_create_file (const char *name, mode_t mode,
                                      struct dentry *parent, void *data,
                                      struct file_operations *fops,
                                      uid_t uid, gid_t gid)
{
        struct dentry *dentry;
        int error;

        dbg("creating file '%s'",name);

        error = fs_create_by_name (name, mode, parent, &dentry);
        if (error) {
                dentry = NULL;
        } else {
                if (dentry->d_inode) {
                        if (data)
                                dentry->d_inode->u.generic_ip = data;
                        if (fops)
                                dentry->d_inode->i_fop = fops;
                        dentry->d_inode->i_uid = uid;
                        dentry->d_inode->i_gid = gid;
                }
        }

        return dentry;
}
```

Figure 4: `fs_create_file` from `drivers/usb/core/inode.c`

which is shown in Figure ??. This function creates a dentry for the file by calling the kernel function `get_dentry()`, and depending on the type of file being created (a file or a directory), either the `usbfs_mkdir()` or `usbfs_create()` function is called. The "normal" VFS functions, `vfs_mkdir()` and `vfs_create()` are not used. If the VFS functions are used, users can create and later delete files, which can lead to confusion, if that is not desired by the filesystem author.

The `struct file_operations` that is needed to create a file, contains the functions that will be called when a user accesses the file. If reading data from the file is all that is desired, then a very simple set of file operations can be specified, as shown by the following code from the `drivers/pci/hotplug/pci_hotplug_core.c` file:

```
/* file ops for the "power" files */
static struct file_operations
    power_file_operations = {
        .read =         power_read_file,
        .write =        power_write_file,
        .open =         default_open,
        .llseek =       generic_file_llseek,
};
```

The `generic_file_llseek()` function lives within the kernel, and provides a default llseek functionality. The `default_open()` function is defined within the driver file as this simple bit of code:

```
static int default_open (
    struct inode *inode, struct file *file)
{
        if (inode->u.generic_ip)
                file->private_data =
                    inode->u.generic_ip;

        return 0;
}
```

This function sets up the file's private data pointer to point to the inode generic data pointer, which was originally the data blob passed to the `fs_create_file()` function. This allows the read and write functions to know what type of device this file is being called for.

The `power_read_file()` and `power_write_file()` functions are called whenever the file is read from or written to. These are also not very complicated functions. For the `power_read_file()` function, the data to return to the user is the current

```
static int fs_create_by_name (const char *name, mode_t mode,
                              struct dentry *parent, struct dentry **dentry)
{
        int error = 0;

        /* If the parent is not specified, we create it in the root.
         * We need the root dentry to do this, which is in the super
         * block. A pointer to that is in the struct vfsmount that we
         * have around.
         */
        if (!parent ) {
                if (usbfs_mount && usbfs_mount->mnt_sb) {
                        parent = usbfs_mount->mnt_sb->s_root;
                }
        }

        if (!parent) {
                dbg("Ah! can not find a parent!");
                return -EFAULT;
        }

        *dentry = NULL;
        down(&parent->d_inode->i_sem);
        *dentry = get_dentry (parent, name);
        if (!IS_ERR(dentry)) {
                if ((mode & S_IFMT) == S_IFDIR)
                        error = usbfs_mkdir (parent->d_inode, *dentry, mode);
                else
                        error = usbfs_create (parent->d_inode, *dentry, mode);
        } else
                error = PTR_ERR(dentry);

        up(&parent->d_inode->i_sem);

        return error;
}
```

Figure 5: `fs_create_by_name` from `drivers/usb/core/inode.c`

power status of a specific PCI slot. The code to do this can be seen in Figure ??. The function allocates a chunk of memory (one page), gets the power status of a specific PCI slot (through the call to `get_power_status()`, and then writes to the chunk of memory, a string representation of this status. The chunk of memory is then copied into user space. This is necessary as the original memory memory is located in kernel space, and needs to be copied to user space into the buffer passed from the user to the `read(2)` call. So when a user issues the command:

```
cat /proc/bus/pci/slots/slot2/power
```

the result is:

```
1
```

The `power_write_file()` function is just as simple and can be seen in Figure ??. With this function the user is able to control the power of the pci slot with a simple`echo(1)` command like:

```
echo 1 > /proc/bus/pci/slots/slot3/power
```

to turn on the power to the third PCI slot in the system.

With those two simple functions, a user can interact with a driver without making special `ioctl(2)` calls.

8 Shutting down

When the driver shuts down it must remove all of the files that it had originally created in the filesystem, in order to be allowed to unmount the filesystem, and free up all of the allocated memory. An example of how to do this can be seen in the `fs_remove_file()` function in the `drivers/usb/core/inode.c` file:

```
static void fs_remove_file (struct dentry *dentry)
{
    struct dentry *parent = dentry->d_parent;

    if (!parent || !parent->d_inode)
        return;

    down(&parent->d_inode->i_sem);
    if (usbfs_positive(dentry)) {
        if (dentry->d_inode) {
            if (S_ISDIR(dentry->d_inode->i_mode))
                usbfs_rmdir(parent->d_inode,
                            dentry);
            else
                usbfs_unlink(parent->d_inode,
                             dentry);
        dput(dentry);
        }
    }
    up(&parent->d_inode->i_sem);
}
```

This function needs a pointer to the dentry that the call to `fs_create_file()` returned. It determines if the dentry has a valid parent as the parent of the dentry is required in order to be able to remove it. Then it calls either the `usbfs_rmdir()` or `usbfs_unlink()` functions to remove the file, depending on if the file is a directory or a not. Again, if the normal VFS layer functions `vfs_rmdir()` and `vfs_unlink()` were used, then any user with the proper permissions would be able to remove any file in the filesystem, which is not what is usually wanted.

9 sysfs

With the above functions, it is simple to create a filesystem that can be included within a driver. But if a driver only wants to export a few files, and the overhead of a separate filesystem is too great, the `sysfs` filesystem should be used. This filesystem provides a external view of a large majority of all of the kernel's data structures and the relationships between them. It is much easier to use `sysfs` than it is to create a separate filesystem.

For more information on how to use `sysfs`, please see the `Documentation/filesystems/sysfs.txt` file in the kernel tree, and Pat Mochel's paper and presentation at the Linux.conf.au 2003 conference.

10 Acknowledgments

I would like to thank Pat Mochel for writing the `ddfs/driverfs/sysfs` code, upon which a lot of the `pcihpfs` and `usbfs` code was originally based.

I would also like to thank Al Viro for answering a lot of VFS related questions, and for enabling a filesystem to be written with such a small amount of code.

```
static ssize_t power_read_file (struct file *file, char *buf, size_t count, loff_t *offset)
{
        struct hotplug_slot *slot = file->private_data;
        unsigned char *page;
        int retval;
        int len;
        u8 value;

        dbg(" count = %d, offset = %lld\n", count, *offset);

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

        if (slot == NULL) {
                dbg("slot == NULL???\n");
                return -ENODEV;
        }

        page = (unsigned char *)__get_free_page(GFP_KERNEL);
        if (!page)
                return -ENOMEM;

        retval = get_power_status (slot, &value);
        if (retval)
                goto exit;
        len = sprintf (page, "%d\n", value);

        if (copy_to_user (buf, page, len)) {
                retval = -EFAULT;
                goto exit;
        }
        *offset += len;
        retval = len;

exit:
        free_page((unsigned long)page);
        return retval;
}
```

Figure 6: `power_read_file` from `drivers/hotplug/pci_hotplug_core.c`

```
static ssize_t power_write_file (struct file *file, const char *ubuff, size_t count, loff_t *offset)
{
	struct hotplug_slot *slot = file->private_data;
	char *buff;
	unsigned long lpower;
	u8 power;
	int retval = 0;

	if (*offset < 0)
		return -EINVAL;
	if (count == 0 || count > 16384)
		return 0;
	if (*offset != 0)
		return 0;

	if (slot == NULL) {
		dbg("slot == NULL???\n");
		return -ENODEV;
	}

	buff = kmalloc (count + 1, GFP_KERNEL);
	if (!buff)
		return -ENOMEM;
	memset (buff, 0x00, count + 1);

	if (copy_from_user ((void *)buff, (void *)ubuff, count)) {
		retval = -EFAULT;
		goto exit;
	}

	lpower = simple_strtoul (buff, NULL, 10);
	power = (u8)(lpower & 0xff);
	dbg ("power = %d\n", power);

	if (!try_module_get(slot->ops->owner)) {
		retval = -ENODEV;
		goto exit;
	}
	switch (power) {
		case 0:
			if (!slot->ops->disable_slot)
				break;
			retval = slot->ops->disable_slot(slot);
			break;

		case 1:
			if (!slot->ops->enable_slot)
				break;
			retval = slot->ops->enable_slot(slot);
			break;

		default:
			err ("Illegal value specified for power\n");
			retval = -EINVAL;
	}
	module_put(slot->ops->owner);
exit:
	kfree (buff);
	if (retval)
		return retval;
	return count;
}
```

Figure 7: `power_write_file` from `drivers/hotplug/pci_hotplug_core.c`