Operating Systems

Special Devices and File Systems

By Paul Krzyzanowski
last update: March 22, 2012

Introduction

In older operating systems, kernel file structures referenced the underlying file system directly. For example, a UNIX operating system would have its file entry reference inodes within the file system. As operating systems had to support different file system types, that direct approach no longer worked. As we have seen, a layer of abstraction, called the Virtual File System (VFS), was created to provide a separation between generic file operations and the implementation that is specific to a particular file system. This architecture allowed file system types to become modular components that could be added on the fly, with the module's initialization function registering the software as a file system.

VFS enabled a single operating system to support multiple file system types. For example, a Linux system may have a disk formatted with an ext4 file system and still access a UDF-formatted DVD (also known as ISO 13346, Universal Disk format (http://en.wikipedia.org/wiki/Universal_Disk_Format)) as well as a FAT-32 file system on an SDHC flash memory card. VFS also enabled network file systems, where the network file system module makes requessts to a remote server.

What the VFS architecture also enabled is the ability to easily create file system modules for things that aren't traditionally viewed as file systems. We tend to think of file systems as components that present us with the ability to manage a hierarchy of named objects (files) whose contents we can create, modify, and delete. However, these objects need not be traditional files sitting on disks (or flash memory). The contents of files and even the names and directories themselves can be dynamically created by the file system module. For example, a *process file system* can provide you with a file system view of the processes on a system and the ability to get, and in some cases change, the state of any process. We refer to these non-standard file systems as *special file systems*.

Why do this? One of the great things about a hierarchical file system is its name space. We intuitively understand the concept of directories, subdirectories, and files. We can browse directories and open files through graphical interfaces, the command line, shell scripts, and with just about any programming or scripting language. We don't need to rely on knowledge of arcane system calls. More importantly, with a file system interface, we may not even need to create specialized system calls. Why do you need a system call to get the current time when you can just open and read `/dev/time`?

Pseudo devices

While we've been talking about file systems, devices also need not refer to physical devices. Given that a device driver is a module containing a bunch of code that implements what *read* and *write* operations to the "device" mean, we can have it do just about any processing.

Null device

The simplest of these is the **null device**. It is known as `/dev/null` on Unix systems, `\Device\Null` on Windows NT and derivatives, and `NUL` on earlier Windows systems. It is also sometimes referred to as a **bit bucket**. The null device is simply a character device that discards any input given to and report and end-of-file whenever a process tries to read from it.

Zero device

The **zero device** is a simple variation of the null device that, instead of returning an

end-of-file, returns an unlimited number bytes whose value is 0.

On POSIX systems, it is located in `/dev/zero`. One common use of it is to create a file of a certain size containing nothing but zeros. For example, the command:

```
dd if=/dev/zero of=nothing bs=1024 count=1024
```

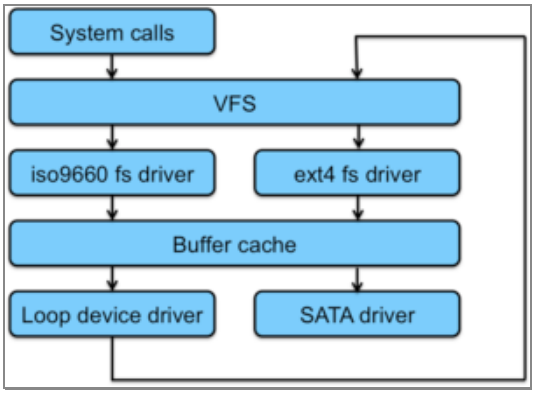
Creates a 1 MB file (block size=1024 × count=1024) named `nothing` filled with zero-value bytes.

Random device

The **random device** is a character device that returns an endless stream of random numbers (bytes). Depending on the operating system and the implementation of the driver, the driver will either use a pseudorandom number generator or provide a high-quality cryptographically secure sequence of random numbers gathered from entropy obtained in various observed events in the system. Most POSIX systems contain two device files: `/dev/random` and `/dev/urandom`. The first is intended for high-quality cryptographically strong random numberes. However, read operations may block until a sufficient amount of events in the system envrionbment are processed to build up large entropy values. The second device, `/dev/urandom`, is non-blocking and may produce more pseudorandom data when collected entropy levels are low.

The loop device

The **loop device** is not a file system. It is a device driver that makes a regular file accessible just like a block device. One associates a specific file with a specific loop device. On Linux, these devices are typically presented with names such as `/dev/loop0`, `/dev/loop1`, etc. The loop device is a block device and provides interfaces to read and write fixed-sized blocks of data. These block read/write requests are transformed by the loop driver into the appropriate read/write requests to the underlying file.



Loop driver

The reason this is useful is that file systems are built on top of block devices. With the loop driver, we can give any file a block device interface and then construct a file system within the file. The common uses for this are to create and access CD or DVD disk images (that may then be burned onto a real disk), distribute software in a way that provides the illusion that it came on a disk image, and also to create encrypted file systems. For this latter case, the loop driver itself often supports encryption so that any file system built on that file will get each data block encrypted and decrypted by the loop driver. Alternatively, one can format a file system that supports encryption onto the file. In either case, this loop device can then be mounted anywhere in the hierarchical name space and that mounted directory can be used for secure storage needs even if the native system fils system is not encrypted.

On Linux, the association between the loop device and file is made via the `losetup` command, which opens the given loop device and uses an *ioctl* system call to send it the name of the file that it should use.

Here's a basic example of using the loop device on Linux:

1. Create a 10 MB file named `file.img`:

```
# dd if=/dev/zero of=file.img bs=1k count=10000
10000+0 records in
```

```
10000+0 records out
```

2. Associate the loop device `/dev/loop0` with the file `file.img`:

```
# losetup /dev/loop0 file.img
```

3. Create an ext2 file system on this loop device (and hence, on `file.img`):

```
mke2fs -c /dev/loop0 10000
mke2fs 1.41.12 (17-May-2010)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
Stride=0 blocks, Stripe width=0 blocks
...
```

4. We now have a file system. Mount it into our name space at `/mnt/here`.

```
# mkdir /mnt/here
# mount -t ext2 /dev/loop0 /mnt/here
```

5. Now we can create and access files and directories under `/mnt/here` and they will all be stored within the file system formatted within the file `file.img`.

Special File Systems

Process file system

The process file system was first created by Tom Killian for the 8th Edition of UNIX. [Versions of UNIX developed in the Computer Science Research Center at Bell Labs were named after the editions of their manuals given that the system was in a continual state of evolution.] It was then considerably refined under in the Plan 9 operating system and given a hierarchy of files per process. Plan 9 was written in part by some of the authors of the original UNIX operating system with an eye to use the things that worked well and dump the rest. One of the things that worked well was the "everything is a file" view of the world.

The process file system (also called *procfs* or *proc file system*) presents a file system name space view of processes and other system information. Under Plan 9, Solaris, and Linux, each process is presented as a directory whose name is the process ID number underneath the mount point `/proc`. Within each process directory, one can read information about the process. For example, `cwd` is a symbolic link to the process' current directory; `cmdline` is a file that contains the command line used to run the process; `stack` is the call stack of the process; `pagemap` contains the page table information for the process; `status` contains status about the process, such as signal maps and context switch counts.

All these files and directories are created dynamically on demand (when accessed). No aspect of the process file system exists in a disk-based file system. procfs is a file system driver that registers itself with VFS and is mounted under `/proc` on the top-level file system. Whenever VFS makes a request to get inodes for files and direcotires, procfs creates them from information it gets by probing various kernel structures, such as the process control block.

In addition to information about processes, procfs has become a conduit to a wide set of information about the running kernel. For example (just a small list):

- `/proc/devices`: enumerates all registered character and block devices.
- `/proc/cpuinfo`: information about the cpu
- `/proc/devices`: list of all registered character and block devices
- `/proc/diskstats`: information about logical disks
- `/proc/meminfo`: information about system and kernel memory

- `/proc/net`: directory containing information about the network protocl stack
- `/proc/swaps`: list of swap partitions
- `/proc/uptime`: time the system has been up
- `/proc/version`: kernel version

This interface provides a richer set of data than is often obtainable through system calls and makes it easier to get this data through file browsing or scripting as well as dedicated programs.

Device file system

One issue with devices under POSIX systems is that they could be located only by name and the names were present in a regular file system (e.g., root disk) as special device files. These files were created with the *mknod* command (which uses a system call by the same name) and contained metadata in the inode that identified them as character or block files. The metadata of the file also identified the major and minor numbers of the corresponding device. This worked well until devices became more dynamic. When bus controllers discovered new devices, such as USB-connected disk drives, we needed to be able to create and delete device files as those devices were discovered or were no longer connected.

One approach to this was to replace the tradition of creating special files in the `/dev` directory and instead create a special-purpose file system, **devfs** that is mounted on `/dev` and presents a file system view of all the devices registed in the kernel. Essentially, it is a view into the kernel's character and block device tables.

The `devfs` file system is still present in Apple's OS X and other systems. It has been evicted from Linux as of the 2.6 kernel and replaced with a `udev` device manager. This device manager is a user-level process that gets a list of existing devices from the kernel and then reads device creation and removal events from the kernel. The udev manager (http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev/udev.html) supports a rich set of rules for creating and deleting device files based on events.

FUSE: User-level file system

The last example of a special purpose file system that we will look at isn't really a file system but rather a framework for file systems. FUSE is a file system module that allows one to create file systems in user space. FUSE contains three parts:

1. The FUSE kernel module: This module registers itself as a file file system with VFS. It serves an a conduit between VFS requests and the user-level interpretation of them.
2. A userspace library (libfuse): This library contains support functions to allow the user file system daemon to interface with the FUSE protocol.
3. A mount utility (fusermount): Mounting file systems requires administrative privileges. Fusermount is a helper program that is installed setuid root. This means that it runs with administrative privileges whenever any user runs it. It allows an ordinary user to create a directory that will serve as a mount point for their user-level file system and mount it via fusermount. The mount starts a file system daemon process (background process) that is run with the user's privileges and in interprets all file system operations that it receives from the kernel module. It communicates with the kernel module via a file descriptor it obtained by opening `/dev/fuse`.

References

- The Design and Implementation of the 4.4BSD Operating System, Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. QuartermanA. ©1996 Addison-Wesley Longman, Inc.
- /dev/random (http://en.wikipedia.org/wiki//dev/random), Wikipedia article.

- Loop device (http://en.wikipedia.org/wiki/Loop_device), Wikipedia article.
- Bell Labs' Plan 9 research project looks to tomorrow (http://doc.cat-v.org/plan_9/1st_edition/designing_plan_9), Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. UKUUG Conference in London, July 1990.
- udev: Linux dynamic device management (http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev/udev.html), kernel.org documentation.
- Hotplugging with udev (http://free-electrons.com/doc/udev.pdf), Michael Opdenacker. Free Electrons, © 2004-2009.
- FUSE: Filesystem in Userspace (http://fuse.sourceforge.net/), Sourcceforge.net.
