Files and the Filesystems

Linux Files

The file is the most basic and fundamental abstraction in Linux. Linux follows the *everything-is-a-file* philosophy. Consequently, much interaction occurs via reading of and writing to files, even when the object in question is not what you would consider a normal file.

In order to be accessed, a file must first be opened. Files can be opened for reading, writing, or both. An open file is referenced via a unique descriptor, a mapping from the metadata associated with the open file back to the specific file itself.

Regular files

What most of us call "files" are what Linux labels *regular files*. A regular file contains bytes of data, organized into a linear array called a byte stream. In Linux, no further organization or formatting is specified for a file. The bytes may have any values, and they may be organized within the file in any way. At the system level, Linux does not enforce a structure upon files beyond the byte stream. Some operating systems, such as VMS, provide highly structured files, supporting concepts such as *records*. Linux does not.

Any of the bytes within a file may be read from or written to. These operations start at a specific byte, which is one's conceptual "location" within the file. This location is called the *file position* or *file offset*. The file position is an essential piece of the metadata that the kernel associates with each open file. When a file is first opened, the file position is zero. Usually, as bytes in the file are read from or written to, byte-by-byte, the file position increases in kind. The file position may also be set manually to a given value, even a value beyond the end of the file. The file position's maximum value is bounded only by the size of the C type used to store it, which is 64 bits on a modern Linux system

The size of a file is measured in bytes and is called its *length*.

A single file can be opened more than once, by a different or even the same process. Each open instance of a file is given a unique file descriptor. User-space programs typically must coordinate amongst themselves to ensure that concurrent file accesses are properly synchronized.

inode

Although files are usually accessed via *filenames*, they actually are not directly associated with such names. Instead, a file is referenced by an *inode* (originally short for *information node*), which is assigned an integer value unique to the filesystem (but not necessarily unique across the whole system). This value is called the *inode number*, often abbreviated as *i-number* or *ino*. An inode stores metadata associated with a file, such as its modification timestamp, owner, type, length, and the location of the file's data—but no filename! The inode is both a physical object, located on disk in Unix-style filesystems, and a conceptual entity, represented by a data structure in the Linux kernel.

Directories and links

Accessing a file via its inode number is cumbersome (and also a potential security hole), so files are always opened from user space by a name, not an inode number. *Directories* are used to provide the names with which to access files. A directory acts as a mapping of human-readable names to inode numbers. A name and inode pair is called a *link*. The physical on-disk form of this mapping—for example, a simple table or a hash —is implemented and managed by the kernel code that supports a given filesystem. Conceptually, a directory is viewed like any normal file, with the difference that it contains only a mapping of names to inodes. The kernel directly uses this mapping to perform name-to-inode resolutions.

When a user-space application requests that a given filename be opened, the kernel opens the directory containing the filename and searches for the given name. From the filename, the kernel obtains the inode number. From the inode number, the inode is found. The inode contains metadata associated with the file, including the on-disk location of the file's data. Initially, there is only one directory on the disk, the *root directory*. This directory is usually denoted by the path /. But, as we all know, there are typically many directories on a system. How does the kernel know *which* directory to look in to find a given filename?

As mentioned previously, directories are much like regular files. Indeed, they even have associated inodes. Consequently, the links inside of directories can point to the inodes of other directories. This means directories can nest inside of other directories, forming a hierarchy of directories. This, in turn, allows for the use of the *pathnames* with which all Unix users are familiar—for example, */home/blackbeard/ concorde.png*.

When the kernel is asked to open a pathname like this, it walks each *directory entry* (called a *dentry* inside of the kernel) in the pathname to find the inode of the next entry. In the preceding example, the kernel starts at /, gets the inode for *home*, goes there, gets the inode for *blackbeard*, runs there, and finally gets the inode for *concorde.png*. This operation is called *directory or pathname resolution*. The Linux kernel also employs a cache, called the *dentry cache*, to store the results of directory resolutions, providing for speedier lookups in the future given temporal locality.

A pathname that starts at the root directory is said to be *fully qualified*, and is called an *absolute pathname*. Some pathnames are not fully qualified; instead, they are provided relative to some other directory. These paths are called *relative pathnames*. When provided with a relative pathname, the kernel begins the pathname resolution in the *current working directory*. Together, the combination of a relative pathname and the current working directory is fully qualified. Although directories are treated like normal files, the kernel does not allow them to be opened and manipulated like regular files. Instead, they must be manipulated using a special set of system calls. These system calls allow for the adding and removing of links, which are the only two sensible operations anyhow. If user space were allowed to manipulate directories without the kernel's mediation, it would be too easy for a single simple error to corrupt the filesystem.

Hard links

Hard links cannot span filesystems because an inode number is meaningless outside of the inode's own filesystem.
Conceptually, nothing covered thus far would prevent multiple names resolving to the same inode. Indeed, this is allowed. When multiple links map different names to the same inode, we call them *hard links*. Hard links allow for complex filesystem structures with multiple pathnames pointing to the same data. The hard links can be in the same directory, or in two or more different directories. In either case, the kernel simply resolves the pathname to the correct inode. For example, a specific inode that points to a specific chunk of data can be hard linked from */home/bluebeard/treasure.txt* and */home/blackbeard/ to_steal.txt*.
Deleting a file involves *unlinking* it from the directory structure, which is done simply by removing its name and inode pair from a directory. Because Linux supports hardlinks, however, the filesystem cannot destroy the inode and its associated data on every unlink operation. What if

another hard link existed elsewhere in the filesystem? To ensure that a file is not destroyed until *all* links to it are removed, each inode contains a *link count* that keeps track of the number of links within the filesystem that point to it. When a pathname is unlinked, the link count is decremented by one; only when it reaches zero are the inode and its associated data actually removed from the filesystem.

Symbolic links

To allow links that can span filesystems, and that are a bit simpler and less transparent, Unix systems also implement *symbolic links* (often shortened to *symlinks*).

Symbolic links look like regular files. A symlink has its own inode and data chunk, which contains the complete pathname of the linked-to file. This means symbolic links can point anywhere, including to files and directories that reside on different file systems, and even to files and directories that do not exist. A symbolic link that points to a nonexistent file is called a *broken link*.

Symbolic links incur more overhead than hard links because resolving a symbolic link effectively involves resolving two files: the symbolic link and then the linked-to file.

Symbolic links are also more opaque than hard links. Using hard links is entirely transparent; in fact, it takes effort to find out that a file is linked more than once! Manipulating symbolic links, on the other hand, requires special system calls. This lack of transparency is often considered a positive, as the link structure is explicitly made plain, with symbolic links acting more as *shortcuts* than as filesystem-internal links.

Special files

Special files are kernel objects that are represented as files. Over the years, Unix systems have supported a handful of different special files. Linux supports four: block device files, character device files, named pipes, and Unix domain sockets. Special files are a way to let certain abstractions fit into the filesystem, continuing the everything-is-a-file paradigm. Linux provides a system call to create a special file. Device access in Unix systems is performed via device files, which act and look like normal files residing on the filesystem. Device files may be opened, read from, and written to, allowing user space to access and manipulate devices (both physical and virtual) on the system. Unix devices are generally broken into two groups: *character devices* and *block devices*. Each type of device has its own special device file.

A character device is accessed as a linear queue of bytes. The device driver places bytes onto the queue, one by one, and user space reads the bytes in the order that they were placed on the queue. A keyboard is an example of a character device. If the user types "peg," for example, an application would want to read from the keyboard device the *p*, the *e*, and, finally, the *g*, in exactly that order. When there are no more characters left to read, the device returns end-of-file (EOF). Missing a character, or reading them in any other order, would make little sense. Character devices are accessed via *character device files*.

A block device, in contrast, is accessed as an array of bytes. The device driver maps the bytes over a seekable device, and user space is free to access any valid bytes in the array, in any order—it might read byte 12, then byte 7, and then byte12 again. Block devices are generally storage devices. Hard disks and flash memory are examples of block devices. They are accessed via *block device files*.

Named pipes (often called *FIFO*s, short for "first in, first out") are an *interprocess communication* (*IPC*) mechanism that provides a communication channel over a file descriptor, accessed via a special file. Regular pipes are the method used to "pipe" the output of one program into the input of another; they are created in memory via a system call and do not exist on any filesystem. Named pipes act like regular pipes but are accessed via a file, called a *FIFO special file*. Unrelated processes can access this file and communicate.

Sockets are the final type of special file. Sockets are an advanced form of IPC that allow for communication between two different processes, not only on the same machine, but even on two different machines. In fact, sockets form the basis of network and Internet programming. They come in multiple varieties, including the Unix domain socket, which is the form of socket used for communication within the local machine. Whereas sockets communicating over the Internet might use a hostname and port pair for identifying the target of communication, Unix domain sockets use a special file residing on a filesystem, often simply called a socket file.

File I/O

Before a file can be read from or written to, it must be opened. The kernel maintains a per-process list of open files, called the *file table*. This table is indexed via nonnegative integers known as *file descriptors* (often abbreviated *fd*s). Each entry in the list contains information about an open file, including a pointer to an in-memory copy of the file's backing inode and associated metadata, such as the file position and access modes. Both user space and kernel space use file descriptors as unique cookies: opening a file returns a file descriptor, while subsequent operations (reading, writing, and so on) take the file descriptor as their primary

argument.
File descriptors are represented by the C int type. Not using a special type is often considered odd, but is, historically, the Unix way. Each Linux process has a maximum number of files that it may open. File descriptors start at 0 and go up to one less than this maximum value. By default, the maximum is 1,024, but it can be configured as high as 1,048,576. Because negative values are not legal file descriptors, −1 is often used to indicate an error from a function that would otherwise return a valid file descriptor.

Unless the process explicitly closes them, every process by convention has at least three file descriptors open: 0, 1, and 2. File descriptor 0 is *standard in* (*stdin*), file descriptor 1 is *standard out* (*stdout*), and file descriptor 2 is *standard error* (*stderr*).Instead of referencing these integers directly, the C library provides the preprocessor defines STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO, respectively. Normally, stdin is connected to the terminal's input device (usually the user's keyboard) and stdout and stderr are connected to the terminal's display. Users can *redirect* these standard file descriptors and even pipe the output of one program into the input of another. This is how the shell implements redirections and pipes.

Opening Files

The most basic method of accessing a file is via the read() and write() system calls. Before a file can be accessed, however, it must be opened via an open() or creat() call. Once done using the file, it should be closed using the system call close().

Permissions of New Files

Both of the previously given forms of the open() system call are valid. The mode argument is ignored unless a file is created; it is required if O_CREAT is given. When a file is created, the mode argument provides the permissions. The mode argument is the familiar Unix permission bitset, such as octal 0644 (owner can read and write, everyone else can only read). Technically speaking, POSIX says the exact values are implementation-specific, allowing different Unix systems to lay out the permission bits however they desired. Every Unix system, however, has implemented the permission bits in the same way. Thus, while technically non-portable, specifying 0644 or 0700 will have the same effect on any system you are likely to come across.

The creat() Function

The combination of O_WRONLY | O_CREAT | O_TRUNC is so common that a system call exists to provide just that behavior:

```
        O_RDONLY, O_WRONLY, or O_RDWR   request opening the
          file read-only, write-only, or read/write, respectively

        O_TRUNC
                    If the file already exists and is a regular file and
the
                    access mode allows writing (i.e., is O_RDWR or
O_WRONLY)
                    it will be truncated to length 0.
```

```
#include <sys/types.h> #include <sys/stat.h> #include <fcntl.h>

int creat (const char *name, mode_t mode);
```

Yes, this function's name is missing an *e*. Ken Thompson, the creator of Unix, once joked that the missing letter was his largest regret in the design of Unix.

The following typical creat() call,

```
int fd;

fd = creat (filename, 0644); if (fd == −1) /* error */
```

is identical to

```
int fd;

fd = open (filename, O_WRONLY | O_CREAT | O_TRUNC, 0644); if (fd == −1) /* error */
```

Reading via read()

The most basic—and common—mechanism used for reading is the read()system call, defined in POSIX.1:

```
#include <unistd.h>

ssize_t read (int fd, void *buf, size_t len);
```

Each call reads up to len bytes into the memory pointed at by buf from the current file offset of the file referenced by fd. On success, the number of bytes written into buf is returned. On error, the call returns −1 and sets errno. The file position is advanced by the number of bytes read from fd. If the object represented by fd is not capable of seeking (for example, a character device file), the read always occurs from the "current" position.

Nonblocking Reads

Sometimes, programmers do not want a call to read() to block when there is no data available. Instead, they prefer that the call return immediately, indicating that no data is available. This is called *nonblocking I/O*; it allows applications to perform I/O, potentially on multiple files, without ever blocking, and thus missing data available in another file.

Writing with write()

The most basic and common system call used for writing is write(). write() is the counterpart of read() and is also defined in POSIX.1:

```
#include <unistd.h>

ssize_t write (int fd, const void *buf, size_t count);
```

A call to write() writes up to count bytes starting at buf to the current position of the file referenced by the file descriptor fd. Files backed by objects that do not support seeking (for example, character devices) always write starting at the "head."

On success, the number of bytes written is returned, and the file position is updated in kind. On error, −1is returned and errno is set appropriately. A call to write()can return 0, but this return value does not have any special meaning; it simply implies that zero bytes were written.

As with read(), the most basic usage is simple:

```
const char *buf = "My ship is solid!"; ssize_t nr;

/* write the string in 'buf' to 'fd' */
nr = write (fd, buf, strlen (buf)); if (nr == −1)
        /* error */
```

Closing Files

After a program has finished working with a file descriptor, it can unmap the file descriptor from the associated file via the close() system call:

```
#include <unistd.h>

int close (int fd);
```

A call to close() unmaps the open file descriptor fd and disassociates the file from the process. The given file descriptor is then no longer valid, and the kernel is free to reuse it as the return value to a subsequent open() or creat() call. A call to close() returns 0 on success. On error, it returns −1 and sets errno appropriately. Usage is simple:

```
if (close (fd) == −1) perror ("close");
```

Filesystems

Filesystems and namespaces

A *filesystem* is a collection of files and directories in a formal and valid hierarchy.

Filesystems usually exist physically (i.e., are stored on disk), although Linux also supports *virtual filesystems* that exist only in memory, and *network filesystems* that exist on machines across the network.

Linux supports a wide range of filesystems—certainly anything that the average user might hope to come across—including media-specific filesystems (for example, ISO9660), network filesystems (*NFS*), native filesystems (*ext4*), filesystems from other Unix systems (*XFS*), and even filesystems from non-Unix systems (*FAT*).

Linux, like all Unix systems, provides a global and unified *namespace* of files and directories. Some operating systems separate different disks and drives into separate namespaces—for example, the hard drive is located at *C:*\. In Unix, that same file might be accessible via the pathname */home/captain/stuff/plank.jpg*, right alongside files from other media. That is, on Unix, the namespace is unified.

Mounting
File-systems may be individually added to and removed from the global namespace of files and directories. These operations are called *mounting* and *unmounting*. Each filesystem is mounted to a specific location in the namespace, known as a *mount point*. The root directory of the filesystem is then accessible at this mount point. For example, a CD might be mounted at */media/cdrom*, making the root of the filesystem on the CD accessible at */media/cdrom*. The first filesystem mounted is located in the root of the namespace, */*, and is called the *root filesystem*. Linux systems always have a root filesystem. Mounting other filesystems at other mount points is optional.

Physical filesystems

Physical filesystems reside on block storage devices, such as CDs, floppy disks, compact flash cards, or hard drives. Some such devices are *partionable*, which means that they can be divided up into multiple filesystems, all of which can be manipulated individually.

The smallest logically addressable unit on a filesystem is the *block*. The block is an abstraction of the filesystem, not of the physical media on which the filesystem resides. A block is usually a power-of-two multiple of the sector size. In Linux, blocks are generally larger than the sector, but they must be smaller than the *page size* (the smallest unit addressable by the *memory management unit*, a hardware component).[4] Common block sizes are 512 bytes, 1 kilobyte, and 4 kilobytes.

The smallest addressable unit on a block device is the *sector*. The sector is a physical attribute of the device. Sectors come in various powers of two, with 512bytes being quite common. A block device cannot transfer or access a unit of data smaller than a sector and all I/O must occur in terms of one or more sectors.

Historically, Unix systems have only a single shared namespace, viewable by all users and all processes on the system. Linux takes an innovative approach and supports *per-process namespaces*, allowing each process to optionally have a unique view of the system's file and directory hierarchy.[5] By default, each process inherits the namespace of its parent, but a process may elect to create its own namespace with its own set of mount points and a unique root directory.

The Virtual Filesystem

The virtual filesystem, occasionally also called a *virtual file switch*, is a mechanism of abstraction that allows the Linux kernel to call filesystem functions and manipulate filesystem data without knowing—or even caring about—the specific type of filesystem being used. The VFS accomplishes this abstraction by providing a *common file model*, which is the basis for all filesystems in Linux. Via function pointers and various object-oriented practices. This approach forces a certain amount of commonality between filesystems. For example, the VFS talks in terms of inodes, superblocks, and directory entries. A filesystem not of Unix origins, possibly devoid of Unix-like concepts such as inodes, simply has to cope. Indeed, cope they do: Linux supports filesystems such as FAT and NTFS without issues. The benefits of the VFS are enormous. A single system call can read from *any* filesystem on *any* medium; a single utility can copy from any one filesystem to any other. All filesystems support the same concepts, the same interfaces, and the same calls. Everything just works—and works well.