CHAPTER 8

File and Directory Management

In Chapters 2, 3, and 4, we covered a multitude of approaches to file I/O. In this chapter, we'll revisit files, this time focusing not on reading from or writing to them, but rather on manipulating and managing them and their metadata.

Files and Their Metadata

As discussed in Chapter 1, each file is referenced by an *inode*, which is addressed by a filesystem-unique numerical value known as an *inode number*. An inode is both a physical object located on the disk of a Unix-style filesystem and a conceptual entity represented by a data structure in the Linux kernel. The inode stores the *metadata* associated with a file, such as the file's access permissions, last access timestamp, owner, group, and size, as well as the location of the file's data.[1]

You can obtain the inode number for a file using the *-i* flag to the *ls* command:

```
$ ls -i
1689459 Kconfig    1689461 main.c     1680144 process.c  1689464 swsusp.c
1680137 Makefile   1680141 pm.c       1680145 smp.c      1680149 user.c
1680138 console.c  1689462 power.h    1689463 snapshot.c
1689460 disk.c     1680143 poweroff.c 1680147 swap.c
```

This output shows that, for example, *disk.c* has an inode number of 1689460. On this particular filesystem, no other file will have this inode number. On a different filesystem, however, we can make no such guarantees.

The Stat Family

Unix provides a family of functions for obtaining the metadata of a file:

1. Interestingly, the one thing not in an inode is the file's name! That's stored in the *directory entry*.

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

int stat (const char *path, struct stat *buf);
int fstat (int fd, struct stat *buf);
int lstat (const char *path, struct stat *buf);
```

Each of these functions returns information about a file. `stat()` returns information about the file denoted by the path `path`, while `fstat()` returns information about the file represented by the file descriptor `fd`. `lstat()` is identical to `stat()`, except that in the case of a symbolic link, `lstat()` returns information about the link itself and not the target file.

Each of these functions stores information in a `stat` structure, which is provided by the user. The `stat` structure is defined in `<bits/stat.h>`, which is included from `<sys/stat.h>`:

```
struct stat {
        dev_t st_dev;         /* ID of device containing file */
        ino_t st_ino;         /* inode number */
        mode_t st_mode;       /* permissions */
        nlink_t st_nlink;     /* number of hard links */
        uid_t st_uid;         /* user ID of owner */
        gid_t st_gid;         /* group ID of owner */
        dev_t st_rdev;        /* device ID (if special file) */
        off_t st_size;        /* total size in bytes */
        blksize_t st_blksize; /* blocksize for filesystem I/O */
        blkcnt_t st_blocks;   /* number of blocks allocated */
        time_t st_atime;      /* last access time */
        time_t st_mtime;      /* last modification time */
        time_t st_ctime;      /* last status change time */
};
```

In more detail, the fields are as follows:

- The `st_dev` field describes the device node on which the file resides (we will cover device nodes later in this chapter). If the file is not backed by a device—for example, if it resides on an NFS volume—this value is `0`.
- The `st_ino` field provides the file's inode number.
- The `st_mode` field provides the file's mode bytes, which describe the file type (such as a regular file or a directory) and the access permissions (such as world-readable). Chapters 1 and 2 covered mode bytes and permissions.
- The `st_nlink` field provides the number of hard links pointing at the file. Every file on a filesystem has at least one hard link.

- The `st_uid` field provides the user ID of the user who owns the file.
- The `st_gid` field provides the group ID of the group who owns the file.
- If the file is a device node, the `st_rdev` field describes the device that this file represents.
- The `st_size` field provides the size of the file, in bytes.
- The `st_blksize` field describes the preferred block size for efficient file I/O. This value (or an integer multiple of it) is the optimal block size for user-buffered I/O (see Chapter 3).
- The `st_blocks` field provides the number of filesystem blocks allocated to the file. This value multiplied by the block size will be smaller than the value provided by `st_size` if the file has holes (that is, if the file is a sparse file).
- The `st_atime` field contains the last *file access time*. This is the most recent time at which the file was accessed (for example, by `read()` or `execle()`).
- The `st_mtime` field contains the last *file modification time*—that is, the last time the file was written to.
- The `st_ctime` field contains the last *file change time*. The field contains the last time that the file's metadata (for example, its owner or permissions) was changed. This is often misunderstood to be the file creation time, which is not preserved on Linux or other Unix-style systems.

On success, all three calls return `0` and store the file's metadata in the provided `stat` structure. On error, they return `-1` and set `errno` to one of the following:

`EACCES`
: The invoking process lacks search permission for one of the directory components of `path` (`stat()` and `lstat()` only).

`EBADF`
: `fd` is invalid (`fstat()` only).

`EFAULT`
: `path` or `buf` is an invalid pointer.

`ELOOP`
: `path` contains too many symbolic links (`stat()` and `lstat()` only).

`ENAMETOOLONG`
: `path` is too long (`stat()` and `lstat()` only).

`ENOENT`
: A component in `path` does not exist (`stat()` and `lstat()` only).

ENOMEM
: There is insufficient memory available to complete the request.

ENOTDIR
: A component in `path` is not a directory (`stat()` and `lstat()` only).

The following program uses `stat()` to retrieve the size of a file provided on the command line:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

int main (int argc, char *argv[])
{
        struct stat sb;
        int ret;

        if (argc < 2) {
                fprintf (stderr,
                         "usage: %s <file>\n", argv[0]);
                return 1;
        }

        ret = stat (argv[1], &sb);
        if (ret) {
                perror ("stat");
                return 1;
        }

        printf ("%s is %ld bytes\n",
                argv[1], sb.st_size);

        return 0;
}
```

Here is the result of running the program on its own source file:

```
$ ./stat stat.c
stat.c is 392 bytes
```

This program, in turn, reports the file type (such as symbolic link or block device node) of the file given by the first argument to the program:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

int main (int argc, char *argv[])
{
        struct stat sb;
```

```
        int ret;

        if (argc < 2) {
                fprintf (stderr,
                         "usage: %s <file>\n", argv[0]);
                return 1;
        }

        ret = stat (argv[1], &sb);
        if (ret) {
                perror ("stat");
                return 1;
        }

        printf ("File type: ");
        switch (sb.st_mode & S_IFMT) {
        case S_IFBLK:
                printf("block device node\n");
                break;
        case S_IFCHR:
                printf("character device node\n");
                break;
        case S_IFDIR:
                printf("directory\n");
                break;
        case S_IFIFO:
                printf("FIFO\n");
                break;
        case S_IFLNK:
                printf("symbolic link\n");
                break;
        case S_IFREG:
                printf("regular file\n");
                break;
        case S_IFSOCK:
                printf("socket\n");
                break;
        default:
                printf("unknown\n");
                break;
        }

        return 0;
}
```

Finally, this snippet uses `fstat()` to check whether an already opened file is on a physical (as opposed to a network) device:

```
/*
 * is_on_physical_device - returns a positive
 * integer if 'fd' resides on a physical device,
 * 0 if the file resides on a nonphysical or
 * virtual device (e.g., on an NFS mount), and
 * -1 on error.
 */
int is_on_physical_device (int fd)
{
        struct stat sb;
        int ret;

        ret = fstat (fd, &sb);
        if (ret) {
                perror ("fstat");
                return -1;
        }

        return gnu_dev_major (sb.st_dev);
}
```

Permissions

While the stat calls can be used to obtain the permission values for a given file, two other system calls set those values:

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

int chmod (const char *path, mode_t mode);
int fchmod (int fd, mode_t mode);
```

Both `chmod()` and `fchmod()` set a file's permissions to `mode`. With `chmod()`, `path` denotes the relative or absolute pathname of the file to modify. For `fchmod()`, the file is given by the file descriptor `fd`.

The legal values for `mode`, represented by the opaque `mode_t` integer type, are the same as those returned by the `st_mode` field in the `stat` structure. Although the values are simple integers, their meanings are specific to each Unix implementation. Consequently, POSIX defines a set of constants that represent the various permissions (see "Permissions of New Files" on page 29 in Chapter 2 for full details). These constants can be binary-ORed together to form the legal values for `mode`. For example, `(S_IRUSR | S_IRGRP)` sets the file's permissions as both owner- and group-readable.

To change a file's permissions, the effective ID of the process calling `chmod()` or `fchmod()` must match the owner of the file, or the process must have the `CAP_FOWNER` capability.

On success, both calls return `0`. On failure, both calls return `-1` and set `errno` to one of the following error values:

EACCES
: The invoking process lacked search permission for a component in `path` (`chmod()` only).

EBADF
: The file descriptor `fd` is invalid (`fchmod()` only).

EFAULT
: `path` is an invalid pointer (`chmod()` only).

EIO
: An internal I/O error occurred on the filesystem. This is a very bad error to encounter; it could indicate a corrupt disk or filesystem.

ELOOP
: The kernel encountered too many symbolic links while resolving `path` (`chmod()` only).

ENAMETOOLONG
: `path` is too long (`chmod()` only).

ENOENT
: `path` does not exist (`chmod()` only).

ENOMEM
: There is insufficient memory available to complete the request.

ENOTDIR
: A component in `path` is not a directory (`chmod()` only).

EPERM
: The effective ID of the invoking process does not match the owner of the file, and the process lacks the `CAP_FOWNER` capability.

EROFS
: The file resides on a read-only filesystem.

This code snippet sets the file *map.png* to owner-readable and -writable:

```
int ret;

/*
 * Set 'map.png' in the current directory to
 * owner-readable and -writable. This is the
 * same as 'chmod 600 ./map.png'.
 */
ret = chmod ("./map.png", S_IRUSR | S_IWUSR);
if (ret)
        perror ("chmod");
```

This code snippet does the same thing, assuming that `fd` represents the open file *map.png*:

```
int ret;

/*
 * Set the file behind 'fd' to owner-readable
 * and -writable.
 */
ret = fchmod (fd, S_IRUSR | S_IWUSR);
if (ret)
        perror ("fchmod");
```

Both `chmod()` and `fchmod()` are available on all modern Unix systems. POSIX requires the former and makes the latter optional.

Ownership

In the `stat` structure, the `st_uid` and `st_gid` fields provide the file's owner and group, respectively. Three system calls allow a user to change those two values:

```
#include <sys/types.h>
#include <unistd.h>

int chown (const char *path, uid_t owner, gid_t group);
int lchown (const char *path, uid_t owner, gid_t group);
int fchown (int fd, uid_t owner, gid_t group);
```

`chown()` and `lchown()` set the ownership of the file specified by the path `path`. They have the same effect, unless the file is a symbolic link: the former follows symbolic links and changes the ownership of the link target rather than the link itself, while `lchown()` does not follow symbolic links and therefore changes the ownership of the symbolic link file instead. `fchown()` sets the ownership of the file represented by the `fd` file descriptor.

On success, all three calls set the file's owner to `owner`, set the file's group to `group`, and return `0`. If either the `owner` or the `group` field is `-1`, that value is not set. Only a process with the `CAP_CHOWN` capability (usually a root process) may change the owner of a file. The owner of a file can change the file's group to any group to which the user is a member; processes with `CAP_CHOWN` can change the file's group to any value.

On failure, the calls return `-1`, and set `errno` to one of the following values:

`EACCES`
: The invoking process lacks search permission for a component in `path` (`chown()` and `lchown()` only).

`EBADF`
: `fd` is invalid (`fchown()` only).

`EFAULT`
: `path` is invalid (`chown()` and `lchown()` only).

`EIO`
: There was an internal I/O error (this is bad).

`ELOOP`
: The kernel encountered too many symbolic links in resolving `path` (`chown()` and `lchown()` only).

`ENAMETOOLONG`
: `path` is too long (`chown()` and `lchown()` only).

`ENOENT`
: The file does not exist.

`ENOMEM`
: There is insufficient memory available to complete the request.

`ENOTDIR`
: A component in `path` is not a directory (`chown()` and `lchown()` only).

`EPERM`
: The invoking process lacked the necessary rights to change the owner or the group as requested.

`EROFS`
: The filesystem is read-only.

This code snippet changes the group of the file *manifest.txt* in the current working directory to *officers*. For this to succeed, the invoking user either must possess the `CAP_CHOWN` capability or must be *kidd* and in the *officers* group:

```
struct group *gr;
int ret;
/*
 * getgrnam() returns information on a group
 * given its name.
 */
gr = getgrnam ("officers");
if (!gr) {
        /* likely an invalid group */
        perror ("getgrnam");
        return 1;
}

/* set manifest.txt's group to 'officers' */
ret = chown("manifest.txt", -1, gr->gr_gid);
if (ret)
        perror ("chown");
```

Before invocation, the file's group is *crew*:

```
$ ls -l
-rw-r--r--  1 kidd  crew  13274 May 23 09:20 manifest.txt
```

After invocation, the file is for the sole privilege of the officers:

```
$ ls -l
-rw-r--r--  1 kidd  officers 13274 May 23 09:20 manifest.txt
```

The file's owner, *kidd*, is not changed because the code snippet passed `-1` for `uid`.

This function sets the file represented by `fd` to root ownership and group:

```
/*
 * make_root_owner - changes the owner and group of the file
 * given by 'fd' to root. Returns 0 on success and -1 on
 * failure.
 */
int make_root_owner (int fd)
{
        int ret;

        /* 0 is both the gid and the uid for root */
        ret = fchown (fd, 0, 0);
        if (ret)
                perror ("fchown");

        return ret;
}
```

The invoking process must have the `CAP_CHOWN` capability. As is par for the course with capabilities, this generally means that it must be owned by root.

Extended Attributes

Extended attributes, also called *xattrs*, provide a mechanism for associating key/value pairs with files. In this chapter, we have already discussed all sorts of key/value metadata associated with files: the file's size, owner, last modification timestamp, and so on. Extended attributes allow existing filesystems to support new features that weren't anticipated in their original designs, such as mandatory access controls for security. What makes extended attributes interesting is that user-space applications may arbitrarily create, read from, and write to the key/value pairs.

Extended attributes are *filesystem-agnostic*, in the sense that applications use a standard interface for manipulating them; the interface is not specific to any filesystem. Applications can thus use extended attributes without concern for what filesystem the files reside on or how the filesystem internally stores the keys and values. Still, the implementation of extended attributes is very filesystem-specific. Different filesystems store

extended attributes in quite different ways, but the kernel hides these differences, abstracting them away behind the extended attribute interface.

The *ext4* filesystem, for example, stores a file's extended attributes in empty space in the file's inode.[2] This feature makes reading extended attributes very fast. Because the filesystem block containing the inode is read off the disk and into memory whenever an application accesses a file, the extended attributes are "automatically" read into memory and can be accessed without any additional overhead.

Other filesystems, such as *FAT* and *minixfs*, do not support extended attributes at all. These filesystems return `ENOTSUP` when extended attribute operations are invoked on their files.

Keys and values

A unique *key* identifies each extended attribute. Keys must be valid UTF-8. They take the form *namespace.attribute*. Every key must be fully qualified; that is, it must begin with a valid namespace, followed by a period. An example of a valid key name is *user.mime_type*; this key is in the *user* namespace with the attribute name *mime_type*.

Old and New Ways for a Filesystem to Store MIME Types

GUI file managers, such as GNOME's, behave differently for files of varying types: they offer unique icons, different default click behavior, special lists of operations to perform, and so on. To accomplish this, the file manager has to know the format of each file. To determine the format, systems such as Windows simply look at the file's extension. For reasons of both tradition and security, however, Unix systems tend to inspect the file and interpret its type. This process is called *MIME type sniffing*.

Some file managers generate this information on the fly; others generate the information once and then cache it. Those that cache the information tend to put it in a custom database. The file manager must work to keep this database in sync with the files, which can change without the file manager's knowledge. A better approach is to jettison the custom database and store such metadata in extended attributes: these are easier to maintain, faster to access, and readily accessible by any application.

A key is either *defined* or *undefined*. If a key is defined, its value may be empty or nonempty. That is, there is a difference between an undefined key and a defined key with no assigned value. As we shall see, this means a special interface is required for removing keys as assigning them an empty value is insufficient.

2. Until the inode runs out of space, of course. Then *ext4* stores extended attributes in additional filesystem blocks.

The value associated with a key, if nonempty, may be any arbitrary array of bytes. Because the value is not necessarily a string, it need not be null-terminated, although null-termination certainly makes sense if you choose to store a C string as a key's value. Since the values are not guaranteed to be null-terminated, all operations on extended attributes require the size of the value. When reading an attribute, the kernel provides the size; when writing an attribute, you must provide the size.

Linux does not enforce any limits on the number of keys, the length of a key, the size of a value, or the total space that can be consumed by all of the keys and values associated with a file. Filesystems, however, have practical limits. These limits are usually manifested as constraints on the total size of all of the keys and values associated with a given file.

With *ext3*, for example, all extended attributes for a given file must fit within the slack space in the file's inode and up to one additional filesystem block. (Older versions of *ext3* were limited to the one filesystem block, without the in-inode storage.) This equates to a practical limit of about 1 KB to 8 KB per file, depending on the size of the filesystem's blocks. *XFS*, in contrast, has no practical limits. Even with *ext3*, these limits are usually not an issue, as most keys and values are short text strings. Nonetheless, keep them in mind—think twice before storing the entire revision control history of a project in a file's extended attributes!

Extended attribute namespaces

The namespaces associated with extended attributes are more than just organizational tools. The kernel enforces different access policies depending on the namespace.

Linux currently defines four extended attribute namespaces and may define more in the future. The current four are as follows:

system
: The *system* namespace is used to implement kernel features that utilize extended attributes, such as access control lists (ACLs). An example of an extended attribute in this namespace is `system.posix_acl_access`. Whether users can read from or write to these attributes depends on the security module in place. Assume at worst that no user (including root) can even read these attributes.

security
: The *security* namespace is used to implement security modules, such as SELinux. Whether user-space applications can access these attributes depends, again, on the security module in place. By default, all processes can read these attributes, but only processes with the `CAP_SYS_ADMIN` capability can write to them.

trusted
: The *trusted* namespace stores restricted information in user space. Only processes with the `CAP_SYS_ADMIN` capability can read from or write to these attributes.

user

The *user* namespace is the standard namespace for use by regular processes. The kernel controls access to this namespace via the normal file permission bits. To read the value from an existing key, a process must have read access to the given file. To create a new key or to write a value to an existing key, a process must have write access to the given file. You can assign extended attributes in the user namespace only to regular files, not to symbolic links or device files. When designing a user-space application that uses extended attributes, this is likely the namespace you want.

Extended Attribute Operations

POSIX defines four operations that applications may perform on a given file's extended attributes:

- Given a file, return a list of all of the file's assigned extended attribute keys.
- Given a file and a key, return the corresponding value.
- Given a file, a key, and a value, assign that value to the key.
- Given a file and a key, remove that extended attribute from the file.

For each operation, POSIX provides three system calls:

- A version that operates on a given pathname; if the path refers to a symbolic link, the target of the link is operated upon (the usual behavior).
- A version that operates on a given pathname; if the path refers to a symbolic link, the link itself is operated upon (the standard `l` variant of a system call).
- A version that operates on a file descriptor (the standard `f` variant).

In the following subsections, we will cover all 12 combinations.

Retrieving an extended attribute

The simplest operation is returning the value of an extended attribute from a file, given the key:

```
#include <sys/types.h>
#include <attr/xattr.h>

ssize_t getxattr (const char *path, const char *key,
                  void *value, size_t size);
ssize_t lgetxattr (const char *path, const char *key,
                   void *value, size_t size);
ssize_t fgetxattr (int fd, const char *key,
                   void *value, size_t size);
```

A successful call to `getxattr()` stores the extended attribute with name `key` from the file `path` in the provided buffer `value`, which is `size` bytes in length. It returns the actual size of the value.

If `size` is `0`, the call returns the size of the value without storing it in `value`. Thus, passing `0` allows applications to determine the correct size for the buffer in which to store the key's value. Given this size, applications can then allocate or resize the buffer as needed.

`lgetxattr()` behaves the same as `getxattr()`, unless `path` is a symbolic link, in which case it returns extended attributes from the link itself rather than the target of the link. Recall from the previous section that attributes in the user namespace cannot be applied to symbolic links—thus, this call is rarely used.

`fgetxattr()` operates on the file descriptor `fd`; otherwise, it behaves the same as `getxattr()`.

On error, all three calls return `-1` and set `errno` to one of the following values:

`EACCES`
: The invoking process lacks search permission for one of the directory components of `path` (`getxattr()` and `lgetxattr()` only).

`EBADF`
: `fd` is invalid (`fgetxattr()` only).

`EFAULT`
: `path`, `key`, or `value` is an invalid pointer.

`ELOOP`
: `path` contains too many symbolic links (`getxattr()` and `lgetxattr()` only).

`ENAMETOOLONG`
: `path` is too long (`getxattr()` and `lgetxattr()` only).

`ENOATTR`
: The attribute `key` does not exist, or the process does not have access to the attribute.

`ENOENT`
: A component in `path` does not exist (`getxattr()` and `lgetxattr()` only).

`ENOMEM`
: There is insufficient memory available to complete the request.

`ENOTDIR`
: A component in `path` is not a directory (`getxattr()` and `lgetxattr()` only).

`ENOTSUP`
: The filesystem on which `path` or `fd` resides does not support extended attributes.

ERANGE
: `size` is too small to hold the value of `key`. As previously discussed, the call may be reissued with `size` set to `0`; the return value will indicate the required buffer size, and `value` may be resized appropriately.

Setting an extended attribute

The following three system calls set a given extended attribute:

```
#include <sys/types.h>
#include <attr/xattr.h>

int setxattr (const char *path, const char *key,
              const void *value, size_t size, int flags);
int lsetxattr (const char *path, const char *key,
               const void *value, size_t size, int flags);
int fsetxattr (int fd, const char *key,
               const void *value, size_t size, int flags);
```

A successful call to `setxattr()` sets the extended attribute `key` on the file `path` to `value`, which is `size` bytes in length. The `flags` field modifies the behavior of the call. If `flags` is `XATTR_CREATE`, the call will fail if the extended attribute already exists. If `flags` is `XATTR_REPLACE`, the call will fail if the extended attribute does not exist. The default behavior, which is performed if `flags` is `0`, allows both creations and replacements. Regardless of the value of `flags`, keys other than `key` are unaffected.

`lsetxattr()` behaves the same as `setxattr()`, unless `path` is a symbolic link, in which case it sets the extended attributes on the link itself, rather than on the target of the link. Recall that attributes in the user namespace cannot be applied to symbolic links—thus, this call is also rarely used.

`fsetxattr()` operates on the file descriptor `fd`; otherwise, it behaves the same as `setxattr()`.

On success, all three system calls return `0`; on failure, the calls return `-1` and set `errno` to one of the following:

EACCES
: The invoking process lacks search permission for one of the directory components of `path` (`setxattr()` and `lsetxattr()` only).

EBADF
: `fd` is invalid (`fsetxattr()` only).

EDQUOT
: A quota limit prevents the space consumption required by the requested operation.

EEXIST
: XATTR_CREATE was set in flags, and key already exists on the given file.

EFAULT
: path, key, or value is an invalid pointer.

EINVAL
: flags is invalid.

ELOOP
: path contains too many symbolic links (setxattr() and lsetxattr() only).

ENAMETOOLONG
: path is too long (setxattr() and lsetxattr() only).

ENOATTR
: XATTR_REPLACE was set in flags, and key does not exist on the given file.

ENOENT
: A component in path does not exist (setxattr() and lsetxattr() only).

ENOMEM
: There is insufficient memory available to complete the request.

ENOSPC
: There is insufficient space on the filesystem to store the extended attribute.

ENOTDIR
: A component in path is not a directory (setxattr() and lsetxattr() only).

ENOTSUP
: The filesystem on which path or fd resides does not support extended attributes.

Listing the extended attributes on a file

The following three system calls enumerate the set of extended attribute keys assigned to a given file:

```
#include <sys/types.h>
#include <attr/xattr.h>

ssize_t listxattr (const char *path,
                   char *list, size_t size);
ssize_t llistxattr (const char *path,
                    char *list, size_t size);
ssize_t flistxattr (int fd,
                    char *list, size_t size);
```

A successful call to `listxattr()` returns a list of the extended attribute keys associated with the file denoted by `path`. The list is stored in the buffer provided by `list`, which is `size` bytes in length. The system call returns the actual size of the list, in bytes.

Each extended attribute key returned in `list` is terminated by a null character, so a list might look like this:

```
"user.md5_sum\0user.mime_type\0system.posix_acl_default\0"
```

Thus, although each key is a traditional, null-terminated C string, you need the length of the entire list (which you can retrieve from the call's return value) to walk the list of keys. To find out how large a buffer you need to allocate, call one of the list functions with a `size` of `0`; this causes the function to return the actual length of the full list of keys. As with `getxattr()`, applications may use this functionality to allocate or resize the buffer to pass for `value`.

`llistxattr()` behaves the same as `listxattr()`, unless `path` is a symbolic link, in which case the call enumerates the extended attribute keys associated with the link itself rather than with the target of the link. Recall that attributes in the user namespace cannot be applied to symbolic links—thus, this call is rarely used.

`flistxattr()` operates on the file descriptor `fd`; otherwise, it behaves the same as `listxattr()`.

On failure, all three calls return `-1` and set `errno` to one of the following error codes:

`EACCES`
: The invoking process lacks search permission for one of the directory components of `path` (`listxattr()` and `llistxattr()` only).

`EBADF`
: `fd` is invalid (`flistxattr()` only).

`EFAULT`
: `path` or `list` is an invalid pointer.

`ELOOP`
: `path` contains too many symbolic links (`listxattr()` and `llistxattr()` only).

`ENAMETOOLONG`
: `path` is too long (`listxattr()` and `llistxattr()` only).

`ENOENT`
: A component in `path` does not exist (`listxattr()` and `llistxattr()` only).

`ENOMEM`
: There is insufficient memory available to complete the request.

ENOTDIR
: A component in path is not a directory (listxattr() and llistxattr() only).

ENOTSUP
: The filesystem on which path or fd resides does not support extended attributes.

ERANGE
: size is nonzero and is insufficiently large to hold the complete list of keys. The application may reissue the call with size set to 0 to discover the actual size of the list. The program may then resize value and reissue the system call.

Removing an extended attribute

Finally, these three system calls remove a given key from a given file:

```
#include <sys/types.h>
#include <attr/xattr.h>

int removexattr (const char *path, const char *key);
int lremovexattr (const char *path, const char *key);
int fremovexattr (int fd, const char *key);
```

A successful call to removexattr() removes the extended attribute key from the file path. Recall that there is a difference between an undefined key and a defined key with an empty (zero-length) value.

lremovexattr() behaves the same as removexattr(), unless path is a symbolic link, in which case the call removes the extended attribute key associated with the link itself rather than with the target of the link. Recall that attributes in the user namespace cannot be applied to symbolic links—thus, this call is also rarely used.

fremovexattr() operates on the file descriptor fd; otherwise, it behaves the same as removexattr().

On success, all three system calls return 0. On failure, all three calls return –1 and set errno to one of the following:

EACCES
: The invoking process lacks search permission for one of the directory components of path (removexattr() and lremovexattr() only).

EBADF
: fd is invalid (fremovexattr() only).

EFAULT
: path or key is an invalid pointer.

`ELOOP`
: `path` contains too many symbolic links (`removexattr()` and `lremovexattr()` only).

`ENAMETOOLONG`
: `path` is too long (`removexattr()` and `lremovexattr()` only).

`ENOATTR`
: `key` does not exist on the given file.

`ENOENT`
: A component in `path` does not exist (`removexattr()` and `lremovexattr()` only).

`ENOMEM`
: There is insufficient memory available to complete the request.

`ENOTDIR`
: A component in `path` is not a directory (`removexattr()` and `lremovexattr()` only).

`ENOTSUP`
: The filesystem on which `path` or `fd` resides does not support extended attributes.

Directories

528.80bIn Unix, a *directory* is a simple concept: it contains a list of filenames, each of which maps to an inode number. Each name is called a *directory entry*, and each name-to-inode mapping is called a *link*. A directory's contents—what the user sees as the result of the *ls* command—are a listing of all the filenames in that directory. When the user opens a file in a given directory, the kernel looks up the filename in that directory's list to find the corresponding inode number. The kernel then passes that inode number to the filesystem, which uses it to find the physical location of the file on the device.

Directories can also contain other directories. A *subdirectory* is a directory inside of another directory. Given this definition, all directories are subdirectories of some *parent directory*, with the exception of the directory at the very root of the filesystem tree, /. Not surprisingly, this directory is called the *root directory* (not to be confused with the root user's home directory, */root*).

A *pathname* consists of a filename along with one or more of its parent directories. An *absolute pathname* is a pathname that begins with the root directory—for example, */usr/bin/sextant*. A *relative pathname* is a pathname that does not begin with the root directory, such as *bin/sextant*. For such a pathname to be useful, the operating system must know the directory to which the path is relative. The current working directory (discussed in the next section) is used as the starting point.

File and directory names can contain any character except /, which delineates directories in a pathname, and *null*, which terminates the pathname. That said, it is standard practice to constrain the characters in pathnames to valid printable characters under the current locale, or even just ASCII. Since neither the kernel nor the C library enforces this practice, however, it is up to applications to enforce the use of only valid printable characters.

Older Unix systems limited filenames to 14 characters. Today, all modern Unix filesystems allow at least 255 bytes for each filename.[3] Many filesystems under Linux allow even longer filenames.[4]

Every directory contains two special directories, . and .. (called *dot* and *dot-dot*). The dot directory is a reference to the directory itself. The dot-dot directory is a reference to the directory's parent directory. For example, */home/kidd/gold/..* is the same directory as */home/kidd*. The root directory's dot and dot-dot directories point to itself—that is, /, /., and /.. are all the same directory. Technically speaking, therefore, one could say that even the root directory is a subdirectory—in this case, of itself.

The Current Working Directory

Every process has a current directory, which it initially inherits from its parent process. That directory is known as the process's *current working directory* (cwd). The current working directory is the starting point from which the kernel resolves relative pathnames. For example, if a process's current working directory is */home/blackbeard*, and that process tries to open *parrot.jpg*, the kernel will attempt to open */home/blackbeard/parrot.jpg*. Conversely, if the process tries to open */usr/bin/mast*, the kernel will indeed open */usr/bin/mast*. The current working directory has no impact on absolute pathnames (that is, pathnames that start with a slash).

A process can both obtain and change its current working directory.

Obtaining the current working directory

The preferred method for obtaining the current working directory is the `getcwd()` system call, which POSIX standardized:

```
#include <unistd.h>

char * getcwd (char *buf, size_t size);
```

3. Note that this limit is 255 *bytes*, not 255 *characters*. Multibyte characters obviously consume more than 1 of these 255 bytes.

4. Of course, older filesystems that Linux provides for backward compatibility, such as FAT, still carry their own limitations. In the case of FAT, this limitation is eight characters, followed by a dot, followed by three characters. Yes, enforcing the dot as a special character inside of the filesystem is silly.

A successful call to `getcwd()` copies the current working directory as an absolute pathname into the buffer pointed at by `buf`, which is of length `size` bytes and returns a pointer to `buf`. On failure, the call returns `NULL` and sets `errno` to one of the following values:

`EFAULT`
: `buf` is an invalid pointer.

`EINVAL`
: `size` is `0`, but `buf` is not `NULL`.

`ENOENT`
: The current working directory is no longer valid. This can happen if the current working directory has been removed.

`ERANGE`
: `size` is too small to hold the current working directory in `buf`. The application needs to allocate a larger buffer and try again.

Here's an example of using `getcwd()`:

```
char cwd[BUF_LEN];

if (!getcwd (cwd, BUF_LEN)) {
        perror ("getcwd");
        exit (EXIT_FAILURE);
}

printf ("cwd = %s\n", cwd);
```

POSIX dictates that the behavior of `getcwd()` is undefined if `buf` is `NULL`. Linux's C library, in this case, will allocate a buffer of length `size` bytes, and store the current working directory there. If `size` is `0`, the C library will allocate a buffer sufficiently large to store the current working directory. It is then the application's responsibility to free the buffer, via `free()`, when it's done with it. Because this behavior is Linux-specific, applications that value portability or a strict adherence to POSIX should not rely on this functionality. This feature, however, does make usage very simple! Here's an example:

```
char *cwd;

cwd = getcwd (NULL, 0);
if (!cwd) {
        perror ("getcwd");
        exit (EXIT_FAILURE);
}

printf ("cwd = %s\n", cwd);

free (cwd);
```

Linux's C library also provides a `get_current_dir_name()` function, which has the same behavior as `getcwd()` when passed a `NULL` `buf` and a `size` of `0`:

```
#define _GNU_SOURCE
#include <unistd.h>

char * get_current_dir_name (void);
```

Thus, this snippet behaves the same as the previous one:

```
char *cwd;

cwd = get_current_dir_name ();
if (!cwd) {
        perror ("get_current_dir_name");
        exit (EXIT_FAILURE);
}

printf ("cwd = %s\n", cwd);

free (cwd);
```

Older BSD systems favored the `getwd()` call, which Linux provides for backward compatibility:

```
#define _XOPEN_SOURCE_EXTENDED /* or _BSD_SOURCE */
#include <unistd.h>

char * getwd (char *buf);
```

A call to `getwd()` copies the current working directory into `buf`, which must be at least `PATH_MAX` bytes in length. The call returns `buf` on success and `NULL` on failure. For example:

```
char cwd[PATH_MAX];

if (!getwd (cwd)) {
        perror ("getwd");
        exit (EXIT_FAILURE);
}

printf ("cwd = %s\n", cwd);
```

For reasons of both portability and security, applications should not use `getwd()`; `getcwd()` is preferred.

Changing the current working directory

When a user first logs into her system, the login process sets her current working directory to her home directory, as specified in */etc/passwd*. Sometimes, however, a process wants to change its current working directory. For example, a shell may want to do this when the user types *cd*.

Linux provides two system calls for changing the current working directory: one that accepts the pathname of a directory and another that accepts a file descriptor representing an open directory.

```
#include <unistd.h>

int chdir (const char *path);
int fchdir (int fd);
```

A call to `chdir()` changes the current working directory to the pathname specified by `path`, which can be an absolute or a relative pathname. Similarly, a call to `fchdir()` changes the current working directory to the pathname represented by the file descriptor `fd`, which must be opened against a directory. On success, both calls return `0`. On failure, both calls return `-1`.

On failure, `chdir()` also sets `errno` to one of the following values:

`EACCES`
: The invoking process lacks search permission for one of the directory components of `path`.

`EFAULT`
: `path` is not a valid pointer.

`EIO`
: An internal I/O error occurred.

`ELOOP`
: The kernel encountered too many symbolic links while resolving `path`.

`ENAMETOOLONG`
: `path` is too long.

`ENOENT`
: The directory pointed at by `path` does not exist.

`ENOMEM`
: There is insufficient memory available to complete the request.

`ENOTDIR`
: One or more of the components in `path` is not a directory.

`fchdir()` sets `errno` to one of the following values:

`EACCES`
: The invoking process lacks search permission for the directory referenced by `fd` (i.e., the execute bit is not set). This happens if the top-level directory is readable but not executable; `open()` succeeds, but `fchdir()` will not.

`EBADF`
: `fd` is not an open file descriptor.

Depending on the filesystem, other error values are valid for either call.

These system calls affect only the currently running process. There is no mechanism in Unix for changing the current working directory of a different process. Therefore, the *cd* command found in shells cannot be a separate process (like most commands) that simply executes `chdir()` on the first command-line argument and then exits. Instead, *cd* must be a special built-in command that causes the shell itself to call `chdir()`, changing its own current working directory.

The most common use of `getcwd()` is to save the current working directory so that the process can return to it later. For example:

```
char *swd;
int ret;

/* save the current working directory */
swd = getcwd (NULL, 0);
if (!swd) {
        perror ("getcwd");
        exit (EXIT_FAILURE);
}

/* change to a different directory */
ret = chdir (some_other_dir);
if (ret) {
        perror ("chdir");
        exit (EXIT_FAILURE);
}

/* do some other work in the new directory... */

/* return to the saved directory */
ret = chdir (swd);
if (ret) {
        perror ("chdir");
        exit (EXIT_FAILURE);
}

free (swd);
```

It's better, however, to `open()` the current directory and then `fchdir()` to it later. This approach is faster because the kernel does not store the pathname of the current working directory in memory; it stores only the inode. Consequently, whenever the user calls `getcwd()`, the kernel must generate the pathname by walking the directory structure. Conversely, opening the current working directory is cheaper because the kernel already has its inode available and the human-readable pathname is not needed to open a file. The following snippet uses this approach:

```
int swd_fd;

swd_fd = open (".", O_RDONLY);
if (swd_fd == -1) {
        perror ("open");
        exit (EXIT_FAILURE);
}

/* change to a different directory */
ret = chdir (some_other_dir);
if (ret) {
        perror ("chdir");
        exit (EXIT_FAILURE);
}

/* do some other work in the new directory... */

/* return to the saved directory */
ret = fchdir (swd_fd);
if (ret) {
        perror ("fchdir");
        exit (EXIT_FAILURE);
}

/* close the directory's fd */
ret = close (swd_fd);
if (ret) {
        perror ("close");
        exit (EXIT_FAILURE);
}
```

This is how shells implement the caching of the previous directory (for example, with *cd -* in *bash*).

A process that does not care about its current working directory—such as a daemon—generally sets it to / with the call `chdir("/")`. An application that interfaces with a user and his data, such as a word processor, generally sets its current working directory to the user's home directory or to a special documents directory. Because current working directories are relevant only in the context of relative pathnames, the current working directory is of most utility to command-line utilities that the user invokes from the shell.

Creating Directories

Linux provides a single system call, standardized by POSIX, for creating new directories:

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

int mkdir (const char *path, mode_t mode);
```

A successful call to `mkdir()` creates the directory `path`, which may be relative or absolute, with the permission bits `mode` (as modified by the current umask), and returns `0`.

The current umask modifies the `mode` argument in the usual way, plus any operating-system-specific mode bits. In Linux, the permission bits of the newly created directory are `(mode & ~umask & 01777)`. In other words, the umask for the process imposes restrictions that the `mkdir()` call cannot override. If the new directory's parent directory has the set group ID (*sgid*) bit set, or if the filesystem is mounted with BSD group semantics, the new directory will inherit the group affiliation from its parent. Otherwise, the effective group ID of the process will apply to the new directory.

On failure, `mkdir()` returns `-1` and sets `errno` to one of the following values:

`EACCES`
: The parent directory is not writable by the current process, or one or more components of `path` are not searchable.

`EEXIST`
: `path` already exists (and not necessarily as a directory).

`EFAULT`
: `path` is an invalid pointer.

`ELOOP`
: The kernel encountered too many symbolic links while resolving `path`.

`ENAMETOOLONG`
: `path` is too long.

`ENOENT`
: A component in `path` does not exist or is a dangling symbolic link.

`ENOMEM`
: There is insufficient kernel memory to complete the request.

`ENOSPC`
: The device containing `path` is out of space, or the user's disk quota is over the limit.

`ENOTDIR`
: One or more of the components in `path` is not a directory.

`EPERM`
: The filesystem containing `path` does not support the creation of directories.

`EROFS`
: The filesystem containing `path` is mounted read-only.

Removing Directories

As the counterpart to `mkdir()`, the POSIX-standardized `rmdir()` removes a directory from the filesystem hierarchy:

```
#include <unistd.h>

int rmdir (const char *path);
```

On success, `rmdir()` removes `path` from the filesystem and returns `0`. The directory specified by `path` must be empty, aside from the dot and dot-dot directories. There is no system call that implements the equivalent of a recursive delete, as with *rm -r*. Such a tool must manually perform a depth-first traversal of the filesystem, removing all files and directories starting with the leaves, and moving back up the filesystem; `rmdir()` can be used at each stage to remove a directory once its files have been removed.

On failure, `rmdir()` returns `-1` and sets `errno` to one of the following values:

`EACCES`
: Write access to the parent directory of `path` is not allowed, or one of the component directories of `path` is not searchable.

`EBUSY`
: `path` is currently in use by the system and cannot be removed. In Linux, this can happen only if `path` is a mount point or a root directory (root directories need not be mount points, thanks to `chroot()`!).

`EFAULT`
: `path` is not a valid pointer.

`EINVAL`
: `path` has the dot directory as its final component.

`ELOOP`
: The kernel encountered too many symbolic links while resolving `path`.

`ENAMETOOLONG`
: `path` is too long.

`ENOENT`
: A component in `path` does not exist, or is a dangling symbolic link.

`ENOMEM`
: There is insufficient kernel memory to complete the request.

`ENOTDIR`
: One or more of the components in `path` is not a directory.

`ENOTEMPTY`
: `path` contains entries other than the special dot and dot-dot directories.

`EPERM`
: The parent directory of `path` has the sticky bit (`S_ISVTX`) set, but the process's effective user ID is neither the user ID of said parent nor of `path` itself, and the process does not have the `CAP_FOWNER` capability. Alternatively, the filesystem containing `path` does not allow the removal of directories.

`EROFS`
: The filesystem containing `path` is mounted read-only.

Usage is simple:

```
int ret;

/* remove the directory /home/barbary/maps */
ret = rmdir ("/home/barbary/maps");
if (ret)
        perror ("rmdir");
```

Reading a Directory's Contents

POSIX defines a family of functions for reading the contents of directories—that is, obtaining a list of the files that reside in a given directory. These functions are useful if you are implementing *ls* or a graphical file save dialog, if you need to operate on every file in a given directory, or if you want to search for files in a directory that match a given pattern.

To begin reading a directory's contents you need to create a *directory stream*, which is represented by a `DIR` object:

```
#include <sys/types.h>
#include <dirent.h>

DIR * opendir (const char *name);
```

A successful call to `opendir()` creates a directory stream representing the directory given by `name`.

A directory stream is little more than a file descriptor representing the open directory, some metadata, and a buffer to hold the directory's contents. Consequently, it is possible to obtain the file descriptor behind a given directory stream:

```
#define _BSD_SOURCE /* or _SVID_SOURCE */
#include <sys/types.h>
#include <dirent.h>

int dirfd (DIR *dir);
```

A successful call to `dirfd()` returns the file descriptor backing the directory stream `dir`. On error, the call returns `-1`. As the directory stream functions use this file descriptor internally, programs should not invoke calls that manipulate the file position. `dirfd()` is a BSD extension and is not standardized by POSIX; programmers wishing to proclaim their POSIX compliance should avoid it.

Reading from a directory stream

Once you have created a directory stream with `opendir()`, your program can begin reading entries from the directory. To do this, use `readdir()`, which returns entries one by one from a given `DIR` object:

```
#include <sys/types.h>
#include <dirent.h>

struct dirent * readdir (DIR *dir);
```

A successful call to `readdir()` returns the next entry in the directory represented by `dir`. The `dirent` structure represents a directory entry. Defined in `<dirent.h>`, on Linux, its definition is:

```
struct dirent {
        ino_t d_ino; /* inode number */
        off_t d_off; /* offset to the next dirent */
        unsigned short d_reclen; /* length of this record */
        unsigned char d_type; /* type of file */
        char d_name[256]; /* filename */
};
```

POSIX requires only the `d_name` field, which is the name of a single file within the directory. The other fields are optional or Linux-specific. Applications desiring portability to other systems, or conformance to POSIX should access only `d_name`.

Applications successively invoke `readdir()`, obtaining each file in the directory, until they find the file they are searching for or until the entire directory is read, at which time `readdir()` returns `NULL`.

On failure, `readdir()` also returns `NULL`. To differentiate between an error and having read all of the files, applications must set `errno` to `0` before each `readdir()` invocation and then check both the return value and `errno`. The only `errno` value set by `readdir()` is `EBADF`, signifying that `dir` is invalid. Thus, many applications do not bother to check for errors and assume that `NULL` means that no more files remain.

Closing the directory stream

To close a directory stream opened with `opendir()`, use `closedir()`:

```
#include <sys/types.h>
#include <dirent.h>
```

```
int closedir (DIR *dir);
```

A successful call to `closedir()` closes the directory stream represented by `dir`, including the backing file descriptor, and returns `0`. On failure, the function returns `-1` and sets `errno` to `EBADF`, the only possible error code, signifying that `dir` is not an open directory stream.

The following snippet implements a function, `find_file_in_dir()`, that uses `readdir()` to search a given directory for a given filename. If the file exists in the directory, the function returns `0`. Otherwise, it returns a nonzero value:

```
/*
 * find_file_in_dir - searches the directory 'path' for a
 * file named 'file'.
 *
 * Returns 0 if 'file' exists in 'path' and a nonzero
 * value otherwise.
 */
int find_file_in_dir (const char *path, const char *file)
{
        struct dirent *entry;
        int ret = 1;
        DIR *dir;

        dir = opendir (path);

        errno = 0;
        while ((entry = readdir (dir)) != NULL) {
                if (strcmp(entry->d_name, file) == 0) {
                        ret = 0;
                        break;
                }
        }

        if (errno && !entry)
                perror ("readdir");

        closedir (dir);
        return ret;
}
```

System calls for reading directory contents

The previously discussed functions for reading the contents of directories are standardized by POSIX and provided by the C library. Internally, these functions use one of two system calls, `readdir()` and `getdents()`, which are provided here for completeness:

```
#include <unistd.h>
#include <linux/types.h>
#include <linux/dirent.h>
```

```
#include <linux/unistd.h>
#include <errno.h>

/*
 * Not defined for user space: need to
 * use the _syscall3() macro to access.
 */
int readdir (unsigned int fd,
             struct dirent *dirp,
             unsigned int count);

int getdents (unsigned int fd,
              struct dirent *dirp,
              unsigned int count);
```

You do not want to use these system calls! They are obtuse and not portable. Instead, user-space applications should use the C library's `opendir()`, `readdir()`, and `closedir()` system calls.

Links

Recall from our discussion of directories that each name-to-inode mapping in a directory is called a *link*. Given this simple definition—that a link is essentially just a name in a list (a directory) that points at an inode—there would appear to be no reason why multiple links to the same inode could not exist. That is, a single inode (and thus a single file) could be referenced from, say, both */etc/customs* and */var/run/ledger*.

Indeed, this is the case, with one catch: because links map to inodes, and inode numbers are specific to a particular filesystem, */etc/customs* and */var/run/ledger* must both reside on the same filesystem. Within a single filesystem, there can be a large number of links to any given file. The only limit is in the size of the integer data type used to hold the number of links. Among various links, no one link is the "original" or the "primary" link. All of the links enjoy the same status, pointing at the same file.

We call these types of links *hard links*. Files can have 0, 1, or many links. Most files have a link count of 1—that is, they are pointed at by a single directory entry—but some files have 2 or even more links. Files with a link count of 0 have no corresponding directory entries on the filesystem. When a file's link count reaches 0, the file is marked as free, and its disk blocks are made available for reuse.[5] Such a file, however, remains on the filesystem if a process has the file open. Once no process has the file open, the file is removed.

5. Finding files with a link count of 0, but whose blocks are marked as allocated is a primary job of *fsck*, the filesystem checker. Such a condition can occur when a file is deleted, but remains open, and the system crashes before the file is closed. The kernel is never able to mark the filesystem blocks as free, and thus the discrepancy arises. Journaling filesystems eliminate this type of error.

The Linux kernel implements this behavior by using a link count and a usage count. The *usage count* is a tally of the number of instances where the file is open. A file is not removed from the filesystem until both the link and the usage counts hit 0.

Another type of link, the *symbolic link*, is not a filesystem mapping, but a higher-level pointer that is interpreted at runtime. Such links may span filesystems—we'll look at them shortly.

Hard Links

The `link()` system call, one of the original Unix system calls, and now standardized by POSIX, creates a new link for an existing file:

```
#include <unistd.h>

int link (const char *oldpath, const char *newpath);
```

A successful call to `link()` creates a new link under the path `newpath` for the existing file `oldpath`, and then returns `0`. Upon completion, both `oldpath` and `newpath` refer to the same file—there is, in fact, no way to even tell which was the "original" link.

On failure, the call returns `-1` and sets `errno` to one of the following:

`EACCES`
: The invoking process lacks search permission for a component in `oldpath`, or the invoking process does not have write permission for the directory containing `newpath`.

`EEXIST`
: `newpath` already exists—`link()` will not overwrite an existing directory entry.

`EFAULT`
: `oldpath` or `newpath` is an invalid pointer.

`EIO`
: An internal I/O error occurred (this is bad!).

`ELOOP`
: Too many symbolic links were encountered in resolving `oldpath` or `newpath`.

`EMLINK`
: The inode pointed at by `oldpath` already has the maximum number of links pointing at it.

`ENAMETOOLONG`
: `oldpath` or `newpath` is too long.

`ENOENT`
: A component in `oldpath` or `newpath` does not exist.

`ENOMEM`
: There is insufficient memory available to complete the request.

`ENOSPC`
: The device containing `newpath` has no room for the new directory entry.

`ENOTDIR`
: A component in `oldpath` or `newpath` is not a directory.

`EPERM`
: The filesystem containing `newpath` does not allow the creation of new hard links, or `oldpath` is a directory.

`EROFS`
: `newpath` resides on a read-only filesystem.

`EXDEV`
: `newpath` and `oldpath` are not on the same mounted filesystem. (Linux allows a single filesystem to be mounted in multiple places, but even in this case, hard links cannot be created across the mount points.)

This example creates a new directory entry, *pirate*, that maps to the same inode (and thus the same file) as the existing file *privateer*, both of which are in */home/kidd*:

```
int ret;

/*
 * create a new directory entry,
 * '/home/kidd/privateer', that points at
 * the same inode as '/home/kidd/pirate'
 */
ret = link ("/home/kidd/privateer", /home/kidd/pirate");
if (ret)
        perror ("link");
```

Symbolic Links

Symbolic links, also known as *symlinks* or *soft links*, are similar to hard links in that both point at files in the filesystem. The symbolic link differs, however, in that it is not merely an additional directory entry, but a special type of file altogether. This special file contains the pathname for a *different* file, called the symbolic link's *target*. At runtime, on the fly, the kernel substitutes this pathname for the symbolic link's pathname (unless using the various `l` versions of system calls, such as `lstat()`, which operate on the link itself, and not the target). Thus, whereas one hard link is indistinguishable from another

hard link to the same file, it is easy to tell the difference between a symbolic link and its target file.

A symbolic link may be relative or absolute. It may also contain the special dot directory discussed earlier, referring to the directory in which it is located, or the dot-dot directory, referring to the parent of this directory. These sorts of "relative" symbolic links are quite common and often rather useful.

Soft links, unlike hard links, can span filesystems. They can point anywhere, in fact! Symbolic links can point at files that exist (the common practice) or at nonexistent files. The latter type of link is called a *dangling symlink*. Sometimes, dangling symlinks are unwanted—such as when the target of the link was deleted, but not the symlink—but at other times, they are intentional. Symbolic links can even point at other symbolic links. This can create loops. System calls that deal with symbolic links check for loops by maintaining a maximum traversal depth. If that depth is surpassed, they return `ELOOP`.

The system call for creating a symbolic link is very similar to its hard link cousin:

```
#include <unistd.h>

int symlink (const char *oldpath, const char *newpath);
```

A successful call to `symlink()` creates the symbolic link `newpath` pointing at the target `oldpath`, and then returns `0`.

On error, `symlink()` returns `-1` and sets `errno` to one of the following:

`EACCES`
: The invoking process lacks search permission for a component in `oldpath`, or the invoking process does not have write permission for the directory containing `newpath`.

`EEXIST`
: `newpath` already exists—`symink()` will not overwrite an existing directory entry.

`EFAULT`
: `oldpath` or `newpath` is an invalid pointer.

`EIO`
: An internal I/O error occurred (this is bad!).

`ELOOP`
: Too many symbolic links were encountered in resolving `oldpath` or `newpath`.

`EMLINK`
: The inode pointed at by `oldpath` already has the maximum number of links pointing at it.

`ENAMETOOLONG`
: `oldpath` or `newpath` is too long.

`ENOENT`
: A component in `oldpath` or `newpath` does not exist.

`ENOMEM`
: There is insufficient memory available to complete the request.

`ENOSPC`
: The device containing `newpath` has no room for the new directory entry.

`ENOTDIR`
: A component in `oldpath` or `newpath` is not a directory.

`EPERM`
: The filesystem containing `newpath` does not allow the creation of new symbolic links.

`EROFS`
: `newpath` resides on a read-only filesystem.

This snippet is the same as our previous example, but it creates */home/kidd/pirate* as a symbolic link (as opposed to a hard link) to */home/kidd/privateer*:

```
int ret;

/*
 * create a symbolic link,
 * '/home/kidd/privateer', that
 * points at '/home/kidd/pirate'
 */
ret = symlink ("/home/kidd/privateer", "/home/kidd/pirate");
if (ret)
        perror ("symlink");
```

Unlinking

The converse to linking is unlinking, the removal of pathnames from the filesystem. A single system call, `unlink()`, handles this task:

```
#include <unistd.h>

int unlink (const char *pathname);
```

A successful call to `unlink()` deletes `pathname` from the filesystem and returns `0`. If that name was the last reference to the file, the file is deleted from the filesystem. If, however, a process has the file open, the kernel will not delete the file from the filesystem until that process closes the file. Once no process has the file open, it is deleted.

If `pathname` refers to a symbolic link, the link, not the target, is destroyed.

If `pathname` refers to another type of special file, such as a device, FIFO, or socket, the special file is removed from the filesystem, but processes that have the file open may continue to utilize it.

On error, `unlink()` returns `-1` and sets `errno` to one of the following error codes:

`EACCES`
: The invoking process does not have write permission for the parent directory of `pathname`, or the invoking process does not have search permission for a component in `pathname`.

`EFAULT`
: `pathname` is an invalid pointer.

`EIO`
: An I/O error occurred (this is bad!).

`EISDIR`
: `pathname` refers to a directory.

`ELOOP`
: Too many symbolic links were encountered in traversing `pathname`.

`ENAMETOOLONG`
: `pathname` is too long.

`ENOENT`
: A component in `pathname` does not exist.

`ENOMEM`
: There is insufficient memory available to complete the request.

`ENOTDIR`
: A component in `pathname` is not a directory.

`EPERM`
: The system does not allow the unlinking of files.

`EROFS`
: `pathname` resides on a read-only filesystem.

`unlink()` does not remove directories. For that, applications should use `rmdir()`, which we discussed earlier (see "Removing Directories" on page 267).

To ease the wanton destruction of any type of file, the C language provides the `remove()` function:

```
#include <stdio.h>

int remove (const char *path);
```

A successful call to `remove()` deletes `path` from the filesystem and returns `0`. If `path` is a file, `remove()` invokes `unlink()`; if `path` is a directory, `remove()` calls `rmdir()`.

On error, `remove()` returns `-1` and sets `errno` to any of the valid error codes set by `unlink()` and `rmdir()`, as applicable.

Copying and Moving Files

Two of the most basic file manipulation tasks are copying and moving files, commonly carried out via the *cp* and *mv* commands. At the filesystem level, *copying* is the act of duplicating a given file's contents under a new pathname. This differs from creating a new hard link to the file in that changes to one file will not affect the other—that is, there now exist two distinct copies of the file, under (at least) two different directory entries. *Moving*, conversely, is the act of renaming the directory entry under which a file is located. This action does not result in the creation of a second copy.

Copying

Although it is surprising to some, Unix does not include a system or library call to facilitate the copying of files and directories. Instead, utilities such as *cp* or GNOME's file manager perform these tasks manually.

In copying a file `src` to a file named `dst`, the steps are as follows:

1. Open `src`.
2. Open `dst`, creating it if it does not exist, and truncating it to zero length if it does exist.
3. Read a chunk of `src` into memory.
4. Write the chunk to `dst`.
5. Continue until all of `src` has been read and written to `dst`.
6. Close `dst`.
7. Close `src`.

If copying a directory, the individual directory and any subdirectories are created via `mkdir()`; each file therein is then copied individually.

Moving

Unlike for copying files, Unix does provide a system call for moving files. The ANSI C standard introduced the call for files, and POSIX standardized it for both files and directories:

```
#include <stdio.h>

int rename (const char *oldpath, const char *newpath);
```

A successful call to `rename()` renames the pathname `oldpath` to `newpath`. The file's contents and inode remain the same. Both `oldpath` and `newpath` must reside on the same filesystem;[6] if they do not, the call will fail. Utilities such as *mv* handle this case by resorting to a copy and unlink.

On success, `rename()` returns `0`, and the file once referenced by `oldpath` is now referenced by `newpath`. On failure, the call returns `-1`, does not touch `oldpath` or `newpath`, and sets `errno` to one of the following values:

`EACCES`
: The invoking process lacks write permission for the parent of `oldpath` or `newpath`, search permission for a component of `oldpath` or `newpath`, or write permission for `oldpath` in the case that `oldpath` is a directory. The last case is an issue because `rename()` must update .. in `oldpath` if it is a directory.

`EBUSY`
: `oldpath` or `newpath` is a mount point.

`EFAULT`
: `oldpath` or `newpath` is an invalid pointer.

`EINVAL`
: `newpath` is contained within `oldpath`, and thus, renaming one to the other would make `oldpath` a subdirectory of itself.

`EISDIR`
: `newpath` exists and is a directory, but `oldpath` is not a directory.

`ELOOP`
: In resolving `oldpath` or `newpath`, too many symbolic links were encountered.

`EMLINK`
: `oldpath` already has the maximum number of links to itself, or `oldpath` is a directory, and `newpath` already has the maximum number of links to itself.

6. Although Linux allows you to mount a device at multiple points in the directory structure, you still cannot rename from one of these mount points to another, even though they are backed by the same device.

`ENAMETOOLONG`
: `oldpath` or `newpath` is too long.

`ENOENT`
: A component in `oldpath` or `newpath` does not exist or is a dangling symbolic link.

`ENOMEM`
: There is insufficient kernel memory to complete the request.

`ENOSPC`
: There is insufficient space on the device to complete the request.

`ENOTDIR`
: A component (aside from potentially the final component) in `oldpath` or `newpath` is not a directory, or `oldpath` is a directory, and `newpath` exists but is not a directory.

`ENOTEMPTY`
: `newpath` is a directory and is not empty.

`EPERM`
: At least one of the paths specified in the arguments exists, the parent directory has the sticky bit set, the invoking process's effective user ID is neither the user ID of the file, nor that of the parent, and the process is not privileged.

`EROFS`
: The filesystem is marked read-only.

`EXDEV`
: `oldpath` and `newpath` are not on the same filesystem.

Table 8-1 reviews the results of moving to and from different types of files.

Table 8-1. Effects of moving to and from different types of files

	Destination is a file	Destination is a directory	Destination is a link	Destination does not exist
Source is a file	The destination is overwritten by the source.	Failure with `EISDIR`.	The file is renamed and the destination is overwritten.	The file is renamed.
Source is a directory	Failure with `ENOTDIR`.	The source is renamed as the destination if the destination is empty; failure with `ENOTEMPTY` otherwise.	The directory is renamed, and the destination is overwritten.	The directory is renamed.
Source is a link	The link is renamed and the destination is overwritten.	Failure with `EISDIR`.	The link is renamed and the destination is overwritten.	The link is renamed.

	Destination is a file	Destination is a directory	Destination is a link	Destination does not exist
Source does not exist	Failure with ENOENT.	Failure with ENOENT.	Failure with ENOENT.	Failure with ENOENT.

For all of these cases, regardless of their type, if the source and destination reside on different filesystems, the call fails and returns `EXDEV`.

Device Nodes

Device nodes are special files that allow applications to interface with device drivers. When an application performs the usual Unix I/O—opening, closing, reading, writing, and so on—on a device node, the kernel does not handle those requests as normal file I/O. Instead, the kernel passes such requests to a device driver. The device driver handles the I/O operation and returns the results to the user. Device nodes provide device abstraction so that applications do not need to be familiar with device specifics, or even master special interfaces. Indeed, device nodes are the standard mechanism for accessing hardware on Unix systems. Network devices are the rare exception, and over the course of Unix's history, some have argued that this exception is a mistake. There is, indeed, an elegant beauty in manipulating all of a machine's hardware using `read()`, `write()`, and `mmap()` calls.

How does the kernel identify the device driver to which it should hand off the request? Each device node is assigned two numerical values, called a *major number* and a *minor number*. These major and minor numbers map to a specific device driver loaded into the kernel. If a device node has a major and minor number that do not correspond to a device driver in the kernel—which occasionally happens, for a variety of reasons—an `open()` request on the device node returns `-1` with `errno` set to `ENODEV`. We say that such device nodes front nonexistent devices.

Special Device Nodes

Several device nodes are present on all Linux systems. These device nodes are part of the Linux development environment, and their presence is considered part of the Linux ABI.

The *null device* has a major number of 1 and a minor number of 3. It lives at */dev/null*. The device file should be owned by root and be readable and writable by all users. The kernel silently discards all write requests to the device. All read requests to the file return end-of-file (EOF).

The *zero device* lives at */dev/zero* and has a major of 1 and a minor of 5. Like the null device, the kernel silently discards writes to the zero device. Reading from the device returns an infinite stream of null bytes.

The *full device*, with a major of 1 and a minor of 7, lives at */dev/full*. As with the zero device, read requests return null characters (`\0`). Write requests, however, always trigger the `ENOSPC` error, signifying that the underlying device is full.

These devices have varied purposes. They are useful for testing how an application handles corner and problem cases—a full filesystem, for example. Because the null and zero devices ignore writes, they also provide a no-overhead way to throw away unwanted I/O.

The Random Number Generator

The kernel's random number generators live at */dev/random* and */dev/urandom*. They have a major number of 1 and minor numbers of 8 and 9, respectively.

The kernel's random number generator gathers noise from device drivers and other sources, and the kernel concatenates together and one-way hashes the gathered noise. The result is then stored in an *entropy pool*. The kernel keeps an estimate of the number of bits of entropy in the pool.

Reads from */dev/random* return entropy from this pool. The results are suitable for seeding random number generators, performing key generation, and other tasks that require cryptographically strong entropy.

In theory, an adversary who was able to obtain enough data from the entropy pool *and* successfully break the one-way hash could gain knowledge about the state of the rest of the entropy pool. Although such an attack is currently only a theoretical possibility—no such attacks are publicly known to have occurred—the kernel reacts to this possibility by decrementing its estimate of the amount of entropy in the pool with each read request. If the estimate reaches zero, the read will block until the system generates more entropy, and the entropy estimate is large enough to satisfy the read.

/dev/urandom does not have this property; reads from the device will succeed even if the kernel's entropy estimate is insufficient to complete the request. Since only the most secure of applications—such as the generation of keys for secure data exchange in GNU Privacy Guard—should care about cryptographically strong entropy, most applications should use */dev/urandom* and not */dev/random*. Reads to the latter can potentially block for a very long time if no I/O activity occurs that feeds the kernel's entropy pool. This is not uncommon on diskless, headless servers.

Out-of-Band Communication

The Unix file model is impressive. With only simple read and write operations, Unix abstracts nearly any conceivable act one could perform on an object. Sometimes, however, programmers need to communicate with a file outside of its primary data stream. For example, consider a serial port device. Reading from the device would read from

the hardware on the far end of the serial port; writing to the device would send data to that hardware. How would a process read one of the serial port's special status pins, such as the data terminal ready (DTR) signal? Alternatively, how would a process set the parity of the serial port?

The answer is to use the `ioctl()` system call. `ioctl()`, which stands for *I/O control*, allows for *out-of-band communication*:

```
#include <sys/ioctl.h>

int ioctl (int fd, int request, ...);
```

The system call requires two parameters:

`fd`
: The file descriptor of a file.

`request`
: A special request code value, predefined and agreed upon by the kernel and the process, that denotes what operation to perform on the file referenced by `fd`.

It may also receive one or more untyped optional parameters (usually unsigned integers or pointers) to pass into the kernel.

The following program uses the `CDROMEJECT` request to eject the media tray from a CD-ROM device, which the user provides as the first argument on the program's command line. This program thus functions similarly to the standard *eject* command:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <linux/cdrom.h>
#include <stdio.h>

int main (int argc, char *argv[])
{
        int fd, ret;

        if (argc < 2) {
                fprintf (stderr,
                         "usage: %s <device to eject>\n",
                         argv[0]);
                return 1;
        }
        /*
         * Opens the CD-ROM device, read-only. O_NONBLOCK
         * tells the kernel that we want to open the device
         * even if there is no media present in the drive.
         */
```

```
        fd = open (argv[1], O_RDONLY | O_NONBLOCK);
        if (fd < 0) {
                perror ("open");
                return 1;
        }

        /* Send the eject command to the CD-ROM device. */
        ret = ioctl (fd, CDROMEJECT, 0);
        if (ret) {
                perror ("ioctl");
                return 1;
        }

        ret = close (fd);
        if (ret) {
                perror ("close");
                return 1;
        }

        return 0;
}
```

The `CDROMEJECT` request is a feature of Linux's CD-ROM device driver. When the kernel receives an `ioctl()` request, it finds the filesystem (in the case of real files) or device driver (in the case of device nodes) responsible for the file descriptor provided and passes on the request for handling. In this case, the CD-ROM device driver receives the request and physically ejects the drive.

Later in this chapter, we will look at an `ioctl()` example that uses an optional parameter to return information to the requesting process.

Monitoring File Events

Linux provides an interface, *inotify*, for monitoring files—for example, to see when they are moved, read from, written to, or deleted. Imagine that you are writing a graphical file manager, such as GNOME's file manager. If a file is copied into a directory while the file manager is displaying its contents, the file manager's view of the directory becomes inconsistent.

One solution is to continually reread the contents of the directory, detecting changes and updating the display. This imposes a periodic overhead and is far from an elegant solution. Worse, there is always a race between when a file is removed from or added to the directory, and when the file manager rereads the directory.

With inotify, the kernel can *push* the event to the application the moment it happens. As soon as a file is deleted, the kernel can notify the file manager. The file manager, in response, can immediately remove the deleted file from the graphical display of the directory.

Many other applications are also concerned with file events. Consider a backup utility or a data-indexing tool. inotify allows both of these programs to operate in real time: the moment a file is created, deleted, or written to, the tools can update the backup archive or data index.

inotify replaces *dnotify*, an earlier file-monitoring mechanism with a cumbersome signals-based interface. Applications should always favor inotify over dnotify. inotify, introduced with kernel 2.6.13, is flexible and easy to use because the same operations that programs perform on regular files work with inotify. We cover only inotify in this book.

Initializing inotify

Before a process can use inotify, the process must initialize it. The `inotify_init()` system call initializes inotify and returns a file descriptor representing the initialized instance:

```
#include <sys/inotify.h>

int inotify_init1 (int flags);
```

The `flags` parameter is usually `0`, but may be a bitwise OR of the following flags:

`IN_CLOEXEC`
: Sets close-on-exec on the new file descriptor.

`IN_NONBLOCK`
: Sets `O_NONBLOCK` on the new file descriptor.

On error, `inotify_init1()` returns `-1` and sets `errno` to one of the following codes:

`EMFILE`
: The per-user limit on the maximum number of inotify instances has been reached.

`ENFILE`
: The system-wide limit on the maximum number of file descriptors has been reached.

`ENOMEM`
: There is insufficient memory available to complete the request.

Let's initialize inotify so we can use it in subsequent steps:

```
int fd;

fd = inotify_init1 (0);
if (fd == -1) {
        perror ("inotify_init1");
        exit (EXIT_FAILURE);
}
```

Watches

After a process initializes inotify, it sets up *watches*. A watch, represented by a *watch descriptor*, is a standard Unix path, and an associated *watch mask*, which tells the kernel what events the process is interested in—for example, reads, writes, or both.

inotify can watch both files and directories. If watching a directory, inotify reports events that occur on the directory itself and on any of the files residing in the directory (but not on files in subdirectories of the watched directory—the watch is not recursive).

Adding a new watch

The system call `inotify_add_watch()` adds a watch for the event or events described by `mask` on the file or directory `path` to the inotify instance represented by `fd`:

```
#include <sys/inotify.h>

int inotify_add_watch (int fd,
                       const char *path,
                       uint32_t mask);
```

On success, the call returns a new watch descriptor. On failure, `inotify_add_watch()` returns `-1` and sets `errno` to one of the following:

`EACCES`
: Read access to the file specified by `path` is not permitted. The invoking process must be able to read the file to add a watch to it.

`EBADF`
: The file descriptor `fd` is not a valid inotify instance.

`EFAULT`
: The pointer `path` is not valid.

`EINVAL`
: The watch mask, `mask`, contains no valid events.

`ENOMEM`
: There is insufficient memory available to complete the request.

`ENOSPC`
: The per-user limit on the total number of inotify watches has been reached.

Watch masks

The watch mask is a binary OR of one or more inotify events, which `<inotify.h>` defines:

`IN_ACCESS`
: The file was read from.

`IN_MODIFY`
: The file was written to.

`IN_ATTRIB`
: The file's metadata (for example, the owner, permissions, or extended attributes) was changed.

`IN_CLOSE_WRITE`
: The file was closed and had been open for writing.

`IN_CLOSE_NOWRITE`
: The file was closed and had not been open for writing.

`IN_OPEN`
: The file was opened.

`IN_MOVED_FROM`
: A file was moved away from the watched directory.

`IN_MOVED_TO`
: A file was moved into the watched directory.

`IN_CREATE`
: A file was created in the watched directory.

`IN_DELETE`
: A file was deleted from the watched directory.

`IN_DELETE_SELF`
: The watched object itself was deleted.

`IN_MOVE_SELF`
: The watched object itself was moved.

The following events are also defined, grouping two or more events into a single value:

`IN_ALL_EVENTS`
: All legal events.

`IN_CLOSE`
: All events related to closing (currently, both `IN_CLOSE_WRITE` and `IN_CLOSE_NOWRITE`).

`IN_MOVE`
: All move-related events (currently, both `IN_MOVED_FROM` and `IN_MOVED_TO`).

Now, we can look at adding a new watch to an existing inotify instance:

```
int wd;

wd = inotify_add_watch (fd, "/etc", IN_ACCESS | IN_MODIFY);
if (wd == -1) {
        perror ("inotify_add_watch");
        exit (EXIT_FAILURE);
}
```

This example adds a watch for all reads or writes on the directory */etc*. If any file in */etc* is written to or read from, inotify sends an event to the inotify file descriptor, `fd`, providing the watch descriptor `wd`. Let's look at how inotify represents these events.

inotify Events

The `inotify_event` structure, defined in `<inotify.h>`, represents inotify events:

```
#include <sys/inotify.h>

struct inotify_event {
        int wd;          /* watch descriptor */
        uint32_t mask;   /* mask of events */
        uint32_t cookie; /* unique cookie */
        uint32_t len;    /* size of 'name' field */
        char name[];     /* nul-terminated name */
};
```

`wd` identifies the watch descriptor, as obtained by `inotify_add_watch()`, and `mask` represents the events. If `wd` identifies a directory and one of the watched-for events occurred on a file within that directory, `name` provides the filename relative to the directory. In this case, `len` is nonzero. Note that `len` is *not* the same as the string length of `name`; `name` can have more than one trailing null character that acts as padding to ensure that a subsequent `inotify_event` is properly aligned. Consequently, you must use `len`, and not `strlen()`, to calculate the offset of the next `inotify_event` structure in an array.

Zero-Length Arrays

`name` is an example of a zero-length array. Zero-length arrays, also known as flexible arrays, are a C99 language feature that allow the creation of arrays of variable length. They have one very powerful use: embedding arrays of variable size in structures. You can think of them as pointers whose contents are inlined at the site of the pointer itself.

> Consider the example of inotify: the obvious way to hand the filename back in this structure is to have a name field such as, say, `name[512]`. But there is no maximum filename length across all filesystems. Any value would put a limit on inotify's utility. Moreover, most filenames are very small, so a large buffer would incur a lot of waste for most files. This situation is not uncommon; the classic solution is to make `name` a pointer, dynamically allocate a buffer elsewhere, and point `name` at it. But that won't work for a system call. A zero-length array was the perfect solution.

For example, if `wd` represents */home/kidd* and has a `mask` of `IN_ACCESS`, and the file */home/kidd/canon* is read from, `name` will equal `canon`, and `len` will be at least `6`. Conversely, if we were watching */home/kidd/canon* directly with the same mask, `len` would be `0` and `name` would be zero-length—you must not touch it.

`cookie` is used to link together two related but disjoint events. We will address it in a subsequent section.

Reading inotify events

Obtaining inotify events is easy: you just read from the file descriptor associated with the inotify instance. inotify provides a feature known as *slurping*, which allows you to read multiple events with a single read request—as many as fit in the buffer provided to `read()`. Because of the variable-length `name` field, this is the most common way to read inotify events.

Our previous example instantiated an inotify instance, and added a watch to that instance. Now, let's read pending events:

```
char buf[BUF_LEN] __attribute__((aligned(4)));
ssize_t len, i = 0;

/* read BUF_LEN bytes' worth of events */
len = read (fd, buf, BUF_LEN);

/* loop over every read event until none remain */
while (i < len) {
        struct inotify_event *event =
                (struct inotify_event *) &buf[i];
        printf ("wd=%d mask=%d cookie=%d len=%d dir=%s\n",
                event->wd, event->mask,
                event->cookie, event->len,
                (event->mask & IN_ISDIR) ? "yes" : "no");

        /* if there is a name, print it */
        if (event->len)
                printf ("name=%s\n", event->name);
```

```
        /* update the index to the start of the next event */
        i += sizeof (struct inotify_event) + event->len;
    }
```

Because the inotify file descriptor acts like a regular file, programs can monitor it via `select()`, `poll()`, and `epoll()`. This allows processes to multiplex inotify events with other file I/O from a single thread.

Advanced inotify events

In addition to the standard events, inotify can generate other events:

`IN_IGNORED`
: The watch represented by `wd` has been removed. This can occur because the user manually removed the watch or because the watched object no longer exists. We will discuss this event in a subsequent section.

`IN_ISDIR`
: The affected object is a directory. (If not set, the affected object is a file.)

`IN_Q_OVERFLOW`
: The inotify queue overflowed. The kernel limits the size of the event queue to prevent unbounded consumption of kernel memory. Once the number of pending events reaches one less than the maximum, the kernel generates this event and appends it to the tail of the queue. No further events are generated until the queue is read from, reducing its size below the limit.

`IN_UNMOUNT`
: The device backing the watched object was unmounted. Thus, the object is no longer available; the kernel will remove the watch and generate the `IN_IGNORED` event.

Any watch can generate these events; the user need not set them explicitly.

Programmers must treat `mask` as a bitmask of pending events. Consequently, do *not* check for events using direct tests of equivalence:

```
/* Do NOT do this! */

if (event->mask == IN_MODIFY)
        printf ("File was written to!\n");
else if (event->mask == IN_Q_OVERFLOW)
        printf ("Oops, queue overflowed!\n);
```

Instead, perform bitwise tests:

```
if (event->mask & IN_ACCESS)
        printf ("The file was read from!\n");
if (event->mask & IN_UNMOUNTED)
        printf ("The file's backing device was unmounted!\n);
```

```
if (event->mask & IN_ISDIR)
        printf ("The file is a directory!\n");
```

Linking together move events

The `IN_MOVED_FROM` and `IN_MOVED_TO` events each represent only half of a move: the former represents the removal from a given location, while the latter represents the arrival at a new location. Therefore, to be truly useful to a program that is attempting to intelligently track files as they move around the filesystem (consider an indexer with the intention that it not reindex moved files), processes need to be able to link the two move events together.

Enter the `cookie` field in the `inotify_event` structure.

The `cookie` field, if nonzero, contains a unique value that links two events together. Consider a process that is watching */bin* and */sbin*. Assume that */bin* has a watch descriptor of 7 and that */sbin* has a watch descriptor of 8. If the file */bin/compass* is moved to */sbin/compass*, the kernel will generate two inotify events.

The first event will have `wd` equal to `7`, `mask` equal to `IN_MOVED_FROM`, and a name of `compass`. The second event will have `wd` equal to `8`, `mask` equal to `IN_MOVED_TO`, and a name of `compass`. In both events, `cookie` will be the same—say, `12`.

If a file is renamed, the kernel still generates two events. `wd` is the same for both.

Note that if a file is moved from or to a directory that is not watched, the process will not receive one of the corresponding events. It is up to the program to notice that the second event with a matching `cookie` never arrives.

Advanced Watch Options

When creating a new watch, you can add one or more of the following values to `mask` to control the behavior of the watch:

`IN_DONT_FOLLOW`
: If this value is set, and if the target of `path` or any of its components is a symbolic link, the link is not followed and `inotify_add_watch()` fails.

`IN_MASK_ADD`
: Normally, if you call `inotify_add_watch()` on a file on which you have an existing watch, the watch mask is updated to reflect the newly provided `mask`. If this flag is set in `mask`, the provided events are *added* to the existing mask.

`IN_ONESHOT`
: If this value is set, the kernel automatically removes the watch after generating the first event against the given object. The watch is, in effect, "one shot."

`IN_ONLYDIR`
: If this value is set, the watch is added only if the object provided is a directory. If `path` represents a file, not a directory, `inotify_add_watch()` fails.

For example, this snippet only adds the watch on */etc/init.d* if *init.d* is a directory and if neither */etc* nor */etc/init.d* is a symbolic link:

```
int wd;

/*
 * Watch '/etc/init.d' to see if it moves, but only if it is a
 * directory and no part of its path is a symbolic link.
 */
wd = inotify_add_watch (fd,
                        "/etc/init.d",
                        IN_MOVE_SELF |
                        IN_ONLYDIR |
                        IN_DONT_FOLLOW);
if (wd == -1)
        perror ("inotify_add_watch");
```

Removing an inotify Watch

As shown in this instance, you can remove a watch from an inotify instance with the system call `inotify_rm_watch()`:

```
#include <inotify.h>

int inotify_rm_watch (int fd, uint32_t wd);
```

A successful call to `inotify_rm_watch()` removes the watch represented by the watch descriptor `wd` from the inotify instance (represented by the file descriptor) `fd` and returns `0`.

For example:

```
int ret;

ret = inotify_rm_watch (fd, wd);
if (ret)
        perror ("inotify_rm_watch");
```

On failure, the system call returns `-1` and sets `errno` to one of the following two options:

`EBADF`
: `fd` is not a valid inotify instance.

`EINVAL`
: `wd` is not a valid watch descriptor on the given inotify instance.

When removing a watch, the kernel generates the `IN_IGNORED` event. The kernel sends this event not only during a manual removal, but when destroying the watch as a side effect of another operation. For example, when a watched file is deleted, any watches on the file are removed. In all such cases, the kernel sends `IN_IGNORED`. This behavior allows applications to consolidate their handling of watch removal in a single place: the event handler for `IN_IGNORED`. This is useful for advanced consumers of inotify that manage complex data structures backing each inotify watch, such as GNOME's Beagle search infrastructure.

Obtaining the Size of the Event Queue

The size of the pending event queue can be obtained via the `FIONREAD` ioctl on the inotify instance's file descriptor. The first argument to the request receives the size of the queue in bytes, as an unsigned integer:

```
unsigned int queue_len;
int ret;

ret = ioctl (fd, FIONREAD, &queue_len);
if (ret < 0)
        perror ("ioctl");
else
        printf ("%u bytes pending in queue\n", queue_len);
```

Note that the request returns the size of the queue in bytes, and not the number of events in the queue. A program can estimate the number of events from the number of bytes, using the known size of the `inotify_event` structure (obtained via `sizeof()`) and a guess at the average size of the `name` field. What's more useful is that the number of bytes pending gives the process an ideal size to read.

The header `<sys/ioctl.h>` defines the `FIONREAD` constant.

Destroying an inotify Instance

Destroying an inotify instance, and any associated watches, is as simple as closing the instance's file descriptor:

```
int ret;

/* 'fd' was obtained via inotify_init() */
ret = close (fd);
if (fd == -1)
        perror ("close");
```

Of course, as with any file descriptor, the kernel automatically closes the file descriptor and cleans up the resource when the process exits.