Instead of referencing these integers directly, the C library provides the preprocessor defines STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO.

Note that file descriptors can reference more than just regular files. They are used for accessing device files and pipes, directories and futexes, FIFOs, and sockets—following the everything-is-a-file philosophy, just about anything you can read or write is accessible via a file descriptor.

Opening Files

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

The open( ) System Call

A file is opened, and a file descriptor is obtained with the open( ) system call:

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

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

The open( ) system call maps the file given by the pathname name to a file descriptor, which it returns on success. The file position is set to zero, and the file is opened for access according to the flags given by flags.

Flags for open( )

The flags argument must be one of O_RDONLY, O_WRONLY, or O_RDWR. Respectively, these arguments request that the file be opened only for reading, only for writing, or for both reading and writing.

For example, the following code opens */home/kidd/madagascar* for reading:

```
int fd;

fd = open ("/home/kidd/madagascar", O_RDONLY);
if (fd == -1)
        /* error */
```

A file opened only for writing *cannot* also be read, and vice versa. The process issuing the open( ) system call must have sufficient permissions to obtain the access requested.

The flags argument can be bitwise-ORed with one or more of the following values, modifying the behavior of the open request:

`O_APPEND`
: The file will be opened in *append mode*. That is, before each write, the file position will be updated to point to the end of the file. This occurs even if another process has written to the file after the issuing process' last write, thereby changing the file position. (See "Append Mode" later in this chapter).

`O_ASYNC`
: A signal (`SIGIO` by default) will be generated when the specified file becomes readable or writable. This flag is available only for terminals and sockets, not for regular files.

`O_CREAT`
: If the file denoted by `name` does not exist, the kernel will create it. If the file already exists, this flag has no effect unless `O_EXCL` is also given.

`O_DIRECT`
: The file will be opened for direct I/O (see "Direct I/O" later in this chapter).

`O_DIRECTORY`
: If `name` is not a directory, the call to `open( )` will fail. This flag is used internally by the `opendir( )` library call.

`O_EXCL`
: When given with `O_CREAT`, this flag will cause the call to `open( )` to fail if the file given by `name` already exists. This is used to prevent race conditions on file creation.

`O_LARGEFILE`
: The given file will be opened using 64-bit offsets, allowing files larger than two gigabytes to be opened. This is implied on 64-bit architectures.

`O_NOCTTY`
: If the given `name` refers to a terminal device (say, */dev/tty*), it will not become the process' controlling terminal, even if the process does not currently have a controlling terminal. This flag is not frequently used.

`O_NOFOLLOW`
: If `name` is a symbolic link, the call to `open( )` will fail. Normally, the link is resolved, and the target file is opened. If other components in the given path are links, the call will still succeed. For example, if `name` is */etc/ship/plank.txt*, the call will fail if *plank.txt* is a symbolic link. It will succeed, however, if *etc* or *ship* is a symbolic link, so long as *plank.txt* is not.

`O_NONBLOCK`
: If possible, the file will be opened in nonblocking mode. Neither the `open( )` call, nor any other operation will cause the process to block (sleep) on the I/O. This behavior may be defined only for FIFOs.

`O_SYNC`
: The file will be opened for synchronous I/O. No write operation will complete until the data has been physically written to disk; normal read operations are already synchronous, so this flag has no effect on reads. POSIX additionally defines `O_DSYNC` and `O_RSYNC`; on Linux, these flags are synonymous with `O_SYNC`. (See "The O_SYNC Flag," later in this chapter.)

`O_TRUNC`
: If the file exists, it is a regular file, and the given flags allow for writing, the file will be truncated to zero length. Use of `O_TRUNC` on a FIFO or terminal device is ignored. Use on other file types is undefined. Specifying `O_TRUNC` with `O_RDONLY` is also undefined, as you need write access to the file in order to truncate it.

For example, the following code opens for writing the file */home/teach/pearl*. If the file already exists, it will be truncated to a length of zero. Because the `O_CREAT` flag is not specified, if the file does not exist, the call will fail:

```
int fd;

fd = open ("/home/teach/pearl", O_WRONLY | O_TRUNC);
if (fd == -1)
        /* error */
```

Owners of New Files

Determining which user owns a new file is straightforward: the uid of the file's owner is the effective uid of the process creating the file.

Determining the owning group is more complicated. The default behavior is to set the file's group to the effective gid of the process creating the file. This is the System V behavior (the behavioral model for much of Linux), and the standard Linux *modus operandi*.

To be difficult, however, BSD defined its own behavior: the file's group is set to the gid of the parent directory. This behavior is available on Linux via a mount-time option*—it is also the behavior that will occur on Linux by default if the file's parent directory has the set group ID (setgid) bit set. Although most Linux systems will use the System V behavior (where new files receive the gid of the creating process), the possibility of the BSD behavior (where new files receive the gid of the parent directory) implies that code that truly cares needs to manually set the group via the `chown( )` system call (see Chapter 7).

Thankfully, caring about the owning group of a file is uncommon.

* The mount options `bsdgroups` or `sysvgroups`.

Permissions of New Files

Both of the previously given forms of the `open()` system call are valid. The `mode` argument is ignored unless a file is created; it is required if `O_CREAT` is given. If you forget to provide the `mode` argument when using `O_CREAT`, the results are undefined, and often quite ugly—so don't forget!

When a file is created, the `mode` argument provides the permissions of the newly created file. The mode is not checked on this particular open of the file, so you can perform contradictory operations, such as opening the file for writing, but assigning the file read-only permissions.

The `mode` argument is the familiar Unix permission bitset, such as octal 0644 (owner can read and write, everyone else can only read). Technically speaking, POSIX allowed the exact values to be implementation-specific, allowing different Unix systems to lay out the permission bits however they desired. To compensate for the nonportability of bit positions in the mode, POSIX introduced the following set of constants that may be binary-ORed together, and supplied for the `mode` argument:

`S_IRWXU`
: Owner has read, write, and execute permission.

`S_IRUSR`
: Owner has read permission.

`S_IWUSR`
: Owner has write permission.

`S_IXUSR`
: Owner has execute permission.

`S_IRWXG`
: Group has read, write, and execute permission.

`S_IRGRP`
: Group has read permission.

`S_IWGRP`
: Group has write permission.

`S_IXGRP`
: Group has execute permission.

`S_IRWXO`
: Everyone else has read, write, and execute permission.

`S_IROTH`
: Everyone else has read permission.

S_IWOTH
: Everyone else has write permission.

S_IXOTH
: Everyone else has execute permission.

The actual permission bits that hit the disk are determined by binary-ANDing the mode argument with the complement of the user's *file creation mask* (*umask*). Informally, the bits in the umask are turned *off* in the mode argument given to open( ). Thus, the usual umask of 022 would cause a mode argument of 0666 to become 0644 (0666 & ~022). As a system programmer, you normally do not take into consideration the umask when setting permissions—the umask exists to allow the user to limit the permissions that his programs set on new files.

As an example, the following code opens the file given by file for writing. If the file does not exist, assuming a umask of 022, it is created with the permissions 0644 (even though the mode argument specifies 0664). If it does exist, it is truncated to zero length:

```
int fd;

fd = open (file, O_WRONLY | O_CREAT | O_TRUNC,
           S_IWUSR | S_IRUSR | S_IWGRP | S_IRGRP | S_IROTH);
if (fd == -1)
        /* error */
```

The creat( ) Function

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

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

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

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

The following typical creat( ) call:

```
int fd;

fd = creat (file, 0644);
if (fd == -1)
        /* error */
```

is identical to:

```
int fd;

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

On most Linux architectures,* `creat( )` is a system call, even though it can be implemented in user space as simply:

```
int creat (const char *name, int mode)
{
        return open (name, O_WRONLY | O_CREAT | O_TRUNC, mode);
}
```

This duplication is a historic relic from when `open( )` had only two arguments. Today, the `creat( )` system call remains around for compatibility. New architectures can implement `creat( )` as shown in *glibc*.

Return Values and Error Codes

Both `open( )` and `creat( )` return a file descriptor on success. On error, both return `-1`, and set `errno` to an appropriate error value (Chapter 1 discussed `errno` and listed the potential error values). Handling an error on file open is not complicated, as generally there will have been few or no steps performed prior to the open that need to be undone. A typical response would be prompting the user for a different filename or simply terminating the program.

Reading via read( )

Now that you know how to open a file, let's look at how to read it. In the following section, we will examine writing.

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

```
#include <unistd.h>

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

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

* Recall that system calls are defined on a per-architecture basis. Thus, while i386 has a `creat( )` system call, Alpha does not. You can use `creat( )` on any architecture, of course, but it may be a library function instead of having its own system call.