CHAPTER

10

System-Level I/O

Input/output (I/O) is the process of copying data between main memory and external devices such as disk drives, terminals, and networks. An input operation copies data from an I/O device to main memory, and an output operation copies data from memory to a device.

All language run-time systems provide higher-level facilities for performing I/O. For example, ANSI C provides the *standard I/O* library, with functions such as `printf` and `scanf` that perform buffered I/O. The C++ language provides similar functionality with its overloaded << ("put to") and >> ("get from") operators. On Linux systems, these higher-level I/O functions are implemented using system-level *Unix I/O* functions provided by the kernel. Most of the time, the higher-level I/O functions work quite well and there is no need to use Unix I/O directly. So why bother learning about Unix I/O?

- *Understanding Unix I/O will help you understand other systems concepts.* I/O is integral to the operation of a system, and because of this, we often encounter circular dependencies between I/O and other systems ideas. For example, I/O plays a key role in process creation and execution. Conversely, process creation plays a key role in how files are shared by different processes. Thus, to really understand I/O, you need to understand processes, and vice versa. We have already touched on aspects of I/O in our discussions of the memory hierarchy, linking and loading, processes, and virtual memory. Now that you have a better understanding of these ideas, we can close the circle and delve into I/O in more detail.
- *Sometimes you have no choice but to use Unix I/O.* There are some important cases where using higher-level I/O functions is either impossible or inappropriate. For example, the standard I/O library provides no way to access file metadata such as file size or file creation time. Further, there are problems with the standard I/O library that make it risky to use for network programming.

This chapter introduces you to the general concepts of Unix I/O and standard I/O and shows you how to use them reliably from your C programs. Besides serving as a general introduction, this chapter lays a firm foundation for our subsequent study of network programming and concurrency.

10.1 Unix I/O

A Linux *file* is a sequence of m bytes:

$$B_0, B_1, \ldots, B_k, \ldots, B_{m-1}$$

All I/O devices, such as networks, disks, and terminals, are modeled as files, and all input and output is performed by reading and writing the appropriate files. This elegant mapping of devices to files allows the Linux kernel to export a simple, low-level application interface, known as *Unix I/O*, that enables all input and output to be performed in a uniform and consistent way:

Opening files. An application announces its intention to access an I/O device by asking the kernel to *open* the corresponding file. The kernel returns a small nonnegative integer, called a *descriptor*, that identifies the file in all subsequent operations on the file. The kernel keeps track of all information about the open file. The application only keeps track of the descriptor.

Each process created by a Linux shell begins life with three open files: *standard input* (descriptor 0), *standard output* (descriptor 1), and *standard error* (descriptor 2). The header file `<unistd.h>` defines constants `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`, which can be used instead of the explicit descriptor values.

Changing the current file position. The kernel maintains a *file position* k, initially 0, for each open file. The file position is a byte offset from the beginning of a file. An application can set the current file position k explicitly by performing a *seek* operation.

Reading and writing files. A *read* operation copies $n > 0$ bytes from a file to memory, starting at the current file position k and then incrementing k by n. Given a file with a size of m bytes, performing a read operation when $k \geq m$ triggers a condition known as *end-of-file (EOF)*, which can be detected by the application. There is no explicit "EOF character" at the end of a file.

Similarly, a *write* operation copies $n > 0$ bytes from memory to a file, starting at the current file position k and then updating k.

Closing files. When an application has finished accessing a file, it informs the kernel by asking it to *close* the file. The kernel responds by freeing the data structures it created when the file was opened and restoring the descriptor to a pool of available descriptors. When a process terminates for any reason, the kernel closes all open files and frees their memory resources.

10.2 Files

Each Linux file has a *type* that indicates its role in the system:

- A *regular file* contains arbitrary data. Application programs often distinguish between *text files*, which are regular files that contain only ASCII or Unicode characters, and *binary files*, which are everything else. To the kernel there is no difference between text and binary files.

  A Linux text file consists of a sequence of *text lines*, where each line is a sequence of characters terminated by a *newline* character ('`\n`'). The newline character is the same as the ASCII line feed character (LF) and has a numeric value of `0x0a`.
- A *directory* is a file consisting of an array of *links*, where each link maps a *filename* to a file, which may be another directory. Each directory contains at

Aside End of line (EOL) indicators

One of the clumsy aspects of working with text files is that different systems use different characters to mark the end of a line. Linux and Mac OS X use '`\n`' (`0xa`), which is the ASCII line feed (LF) character. However, MS Windows and Internet protocols such as HTTP use the sequence '`\r\n`' (`0xd 0xa`), which is the ASCII carriage return (CR) character followed by a line feed (LF). If you create a file `foo.txt` in Windows and then view it in a Linux text editor, you'll see an annoying `^M` at the end of each line, which is how Linux tools display the CR character. You can remove these unwanted CR characters from `foo.txt` in place by running the following command:

```
linux> perl -pi -e "s/\r\n/\n/g" foo.txt
```

least two entries: `.` (dot) is a link to the directory itself, and `..` (dot-dot) is a link to the *parent directory* in the directory hierarchy (see below). You can create a directory with the `mkdir` command, view its contents with `ls`, and delete it with `rmdir`.

- A *socket* is a file that is used to communicate with another process across a network (Section 11.4).

Other file types include *named pipes*, *symbolic links*, and *character* and *block devices*, which are beyond our scope.

The Linux kernel organizes all files in a single *directory hierarchy* anchored by the *root directory* named `/` (slash). Each file in the system is a direct or indirect descendant of the root directory. Figure 10.1 shows a portion of the directory hierarchy on our Linux system.

As part of its context, each process has a *current working directory* that identifies its current location in the directory hierarchy. You can change the shell's current working directory with the `cd` command.

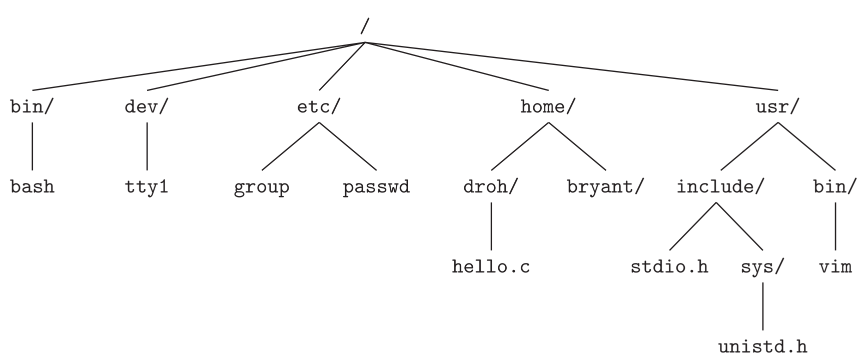

Figure 10.1 **Portion of the Linux directory hierarchy.** A trailing slash denotes a directory.

Locations in the directory hierarchy are specified by *pathnames*. A pathname is a string consisting of an optional slash followed by a sequence of filenames separated by slashes. Pathnames have two forms:

- An *absolute pathname* starts with a slash and denotes a path from the root node. For example, in Figure 10.1, the absolute pathname for `hello.c` is `/home/droh/hello.c`.
- A *relative pathname* starts with a filename and denotes a path from the current working directory. For example, in Figure 10.1, if `/home/droh` is the current working directory, then the relative pathname for `hello.c` is `./hello.c`. On the other hand, if `/home/bryant` is the current working directory, then the relative pathname is `../home/droh/hello.c`.

10.3 Opening and Closing Files

A process opens an existing file or creates a new file by calling the `open` function.

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

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

Returns: new file descriptor if OK, −1 on error

The `open` function converts a `filename` to a file descriptor and returns the descriptor number. The descriptor returned is always the smallest descriptor that is not currently open in the process. The `flags` argument indicates how the process intends to access the file:

O_RDONLY. Reading only

O_WRONLY. Writing only

O_RDWR. Reading and writing

For example, here is how to open an existing file for reading:

```
fd = Open("foo.txt", O_RDONLY, 0);
```

The `flags` argument can also be ORed with one or more bit masks that provide additional instructions for writing:

O_CREAT. If the file doesn't exist, then create a *truncated* (empty) version of it.

O_TRUNC. If the file already exists, then truncate it.

O_APPEND. Before each write operation, set the file position to the end of the file.

Mask	Description
S_IRUSR	User (owner) can read this file
S_IWUSR	User (owner) can write this file
S_IXUSR	User (owner) can execute this file
S_IRGRP	Members of the owner's group can read this file
S_IWGRP	Members of the owner's group can write this file
S_IXGRP	Members of the owner's group can execute this file
S_IROTH	Others (anyone) can read this file
S_IWOTH	Others (anyone) can write this file
S_IXOTH	Others (anyone) can execute this file

Figure 10.2 **Access permission bits.** Defined in `sys/stat.h`.

For example, here is how you might open an existing file with the intent of appending some data:

```
fd = Open("foo.txt", O_WRONLY|O_APPEND, 0);
```

The `mode` argument specifies the access permission bits of new files. The symbolic names for these bits are shown in Figure 10.2.

As part of its context, each process has a `umask` that is set by calling the `umask` function. When a process creates a new file by calling the `open` function with some `mode` argument, then the access permission bits of the file are set to `mode & ~umask`. For example, suppose we are given the following default values for `mode` and `umask`:

```
#define DEF_MODE   S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH
#define DEF_UMASK  S_IWGRP|S_IWOTH
```

Then the following code fragment creates a new file in which the owner of the file has read and write permissions, and all other users have read permissions:

```
umask(DEF_UMASK);
fd = Open("foo.txt", O_CREAT|O_TRUNC|O_WRONLY, DEF_MODE);
```

Finally, a process closes an open file by calling the `close` function.

```
#include <unistd.h>

int close(int fd);
```

Returns: 0 if OK, −1 on error

Closing a descriptor that is already closed is an error.

Practice Problem 10.1 (solution page 951)

What is the output of the following program?

```
#include "csapp.h"

int main()
{
    int fd1, fd2;

    fd1 = Open("foo.txt", O_RDONLY, 0);
    Close(fd1);
    fd2 = Open("baz.txt", O_RDONLY, 0);
    printf("fd2 = %d\n", fd2);
    exit(0);
}
```

10.4 Reading and Writing Files

Applications perform input and output by calling the `read` and `write` functions, respectively.

```
#include <unistd.h>

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

Returns: number of bytes read if OK, 0 on EOF, −1 on error

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

Returns: number of bytes written if OK, −1 on error

The `read` function copies at most `n` bytes from the current file position of descriptor `fd` to memory location `buf`. A return value of −1 indicates an error, and a return value of 0 indicates EOF. Otherwise, the return value indicates the number of bytes that were actually transferred.

The `write` function copies at most `n` bytes from memory location `buf` to the current file position of descriptor `fd`. Figure 10.3 shows a program that uses `read` and `write` calls to copy the standard input to the standard output, 1 byte at a time.

Applications can explicitly modify the current file position by calling the `lseek` function, which is beyond our scope.

In some situations, `read` and `write` transfer fewer bytes than the application requests. Such *short counts* do *not* indicate an error. They occur for a number of reasons:

Aside What's the difference between `ssize_t` and `size_t`?

You might have noticed that the `read` function has a `size_t` input argument and an `ssize_t` return value. So what's the difference between these two types? On x86-64 systems, a `size_t` is defined as an `unsigned long`, and an `ssize_t` (*signed size*) is defined as a `long`. The `read` function returns a signed size rather than an unsigned size because it must return a −1 on error. Interestingly, the possibility of returning a single −1 reduces the maximum size of a `read` by a factor of 2.

code/io/cpstdin.c

```
#include "csapp.h"

int main(void)
{
    char c;

    while(Read(STDIN_FILENO, &c, 1) != 0)
        Write(STDOUT_FILENO, &c, 1);
    exit(0);
}
```

code/io/cpstdin.c

Figure 10.3 Using read and write to copy standard input to standard output 1 byte at a time.

Encountering EOF on reads. Suppose that we are ready to read from a file that contains only 20 more bytes from the current file position and that we are reading the file in 50-byte chunks. Then the next `read` will return a short count of 20, and the `read` after that will signal EOF by returning a short count of 0.

Reading text lines from a terminal. If the open file is associated with a terminal (i.e., a keyboard and display), then each `read` function will transfer one text line at a time, returning a short count equal to the size of the text line.

Reading and writing network sockets. If the open file corresponds to a network socket (Section 11.4), then internal buffering constraints and long network delays can cause `read` and `write` to return short counts. Short counts can also occur when you call `read` and `write` on a Linux *pipe*, an interprocess communication mechanism that is beyond our scope.

In practice, you will never encounter short counts when you read from disk files except on EOF, and you will never encounter short counts when you write to disk files. However, if you want to build robust (reliable) network applications such as Web servers, then you must deal with short counts by repeatedly calling `read` and `write` until all requested bytes have been transferred.

10.5 Robust Reading and Writing with the Rio Package

In this section, we will develop an I/O package, called the Rio (Robust I/O) package, that handles these short counts for you automatically. The Rio package provides convenient, robust, and efficient I/O in applications such as network programs that are subject to short counts. Rio provides two different kinds of functions:

Unbuffered input and output functions. These functions transfer data directly between memory and a file, with no application-level buffering. They are especially useful for reading and writing binary data to and from networks.

Buffered input functions. These functions allow you to efficiently read text lines and binary data from a file whose contents are cached in an application-level buffer, similar to the one provided for standard I/O functions such as `printf`. Unlike the buffered I/O routines presented in [110], the buffered Rio input functions are thread-safe (Section 12.7.1) and can be interleaved arbitrarily on the same descriptor. For example, you can read some text lines from a descriptor, then some binary data, and then some more text lines.

We are presenting the Rio routines for two reasons. First, we will be using them in the network applications we develop in the next two chapters. Second, by studying the code for these routines, you will gain a deeper understanding of Unix I/O in general.

10.5.1 Rio Unbuffered Input and Output Functions

Applications can transfer data directly between memory and a file by calling the `rio_readn` and `rio_writen` functions.

```
#include "csapp.h"

ssize_t rio_readn(int fd, void *usrbuf, size_t n);
ssize_t rio_writen(int fd, void *usrbuf, size_t n);
```

Returns: number of bytes transferred if OK, 0 on EOF (`rio_readn` only), −1 on error

The `rio_readn` function transfers up to `n` bytes from the current file position of descriptor `fd` to memory location `usrbuf`. Similarly, the `rio_writen` function transfers `n` bytes from location `usrbuf` to descriptor `fd`. The `rio_readn` function can only return a short count if it encounters EOF. The `rio_writen` function never returns a short count. Calls to `rio_readn` and `rio_writen` can be interleaved arbitrarily on the same descriptor.

Figure 10.4 shows the code for rio_readn and rio_writen. Notice that each function manually restarts the read or write function if it is interrupted by the return from an application signal handler. To be as portable as possible, we allow for interrupted system calls and restart them when necessary.

10.5.2 Rio Buffered Input Functions

Suppose we wanted to write a program that counts the number of lines in a text file. How might we do this? One approach is to use the read function to transfer 1 byte at a time from the file to the user's memory, checking each byte for the newline character. The disadvantage of this approach is that it is inefficient, requiring a trap to the kernel to read each byte in the file.

A better approach is to call a wrapper function (rio_readlineb) that copies the text line from an internal *read buffer*, automatically making a read call to refill the buffer whenever it becomes empty. For files that contain both text lines and binary data (such as the HTTP responses described in Section 11.5.3), we also provide a buffered version of rio_readn, called rio_readnb, that transfers raw bytes from the same read buffer as rio_readlineb.

```
#include "csapp.h"

void rio_readinitb(rio_t *rp, int fd);
```

Returns: nothing

```
ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n);
```

Returns: number of bytes read if OK, 0 on EOF, −1 on error

The rio_readinitb function is called once per open descriptor. It associates the descriptor fd with a read buffer of type rio_t at address rp.

The rio_readlineb function reads the next text line from file rp (including the terminating newline character), copies it to memory location usrbuf, and terminates the text line with the NULL (zero) character. The rio_readlineb function reads at most maxlen-1 bytes, leaving room for the terminating NULL character. Text lines that exceed maxlen-1 bytes are truncated and terminated with a NULL character.

The rio_readnb function reads up to n bytes from file rp to memory location usrbuf. Calls to rio_readlineb and rio_readnb can be interleaved arbitrarily on the same descriptor. However, calls to these buffered functions should not be interleaved with calls to the unbuffered rio_readn function.

You will encounter numerous examples of the Rio functions in the remainder of this text. Figure 10.5 shows how to use the Rio functions to copy a text file from standard input to standard output, one line at a time.

Figure 10.6 shows the format of a read buffer, along with the code for the rio_readinitb function that initializes it. The rio_readinitb function sets up an empty read buffer and associates an open file descriptor with that buffer.

code/src/csapp.c

```
ssize_t rio_readn(int fd, void *usrbuf, size_t n)
{
    size_t nleft = n;
    ssize_t nread;
    char *bufp = usrbuf;

    while (nleft > 0) {
        if ((nread = read(fd, bufp, nleft)) < 0) {
            if (errno == EINTR) /* Interrupted by sig handler return */
                nread = 0;      /* and call read() again */
            else
                return -1;      /* errno set by read() */
        }
        else if (nread == 0)
            break;              /* EOF */
        nleft -= nread;
        bufp += nread;
    }
    return (n - nleft);         /* Return >= 0 */
}
```

code/src/csapp.c

code/src/csapp.c

```
ssize_t rio_writen(int fd, void *usrbuf, size_t n)
{
    size_t nleft = n;
    ssize_t nwritten;
    char *bufp = usrbuf;

    while (nleft > 0) {
        if ((nwritten = write(fd, bufp, nleft)) <= 0) {
            if (errno == EINTR)  /* Interrupted by sig handler return */
                nwritten = 0;    /* and call write() again */
            else
                return -1;       /* errno set by write() */
        }
        nleft -= nwritten;
        bufp += nwritten;
    }
    return n;
}
```

code/src/csapp.c

Figure 10.4 **The `rio_readn` and `rio_writen` functions.**

code/io/cpfile.c

```
#include "csapp.h"

int main(int argc, char **argv)
{
    int n;
    rio_t rio;
    char buf[MAXLINE];

    Rio_readinitb(&rio, STDIN_FILENO);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0)
        Rio_writen(STDOUT_FILENO, buf, n);
}
```

code/io/cpfile.c

Figure 10.5 Copying a text file from standard input to standard output.

code/include/csapp.h

```
#define RIO_BUFSIZE 8192
typedef struct {
    int rio_fd;                /* Descriptor for this internal buf */
    int rio_cnt;               /* Unread bytes in internal buf */
    char *rio_bufptr;          /* Next unread byte in internal buf */
    char rio_buf[RIO_BUFSIZE]; /* Internal buffer */
} rio_t;
```

code/include/csapp.h

code/src/csapp.c

```
void rio_readinitb(rio_t *rp, int fd)
{
    rp->rio_fd = fd;
    rp->rio_cnt = 0;
    rp->rio_bufptr = rp->rio_buf;
}
```

code/src/csapp.c

Figure 10.6 A read buffer of type `rio_t` and the `rio_readinitb` function that initializes it.

The heart of the Rio read routines is the `rio_read` function shown in Figure 10.7. The `rio_read` function is a buffered version of the Linux `read` function. When `rio_read` is called with a request to read `n` bytes, there are `rp->rio_cnt` unread bytes in the read buffer. If the buffer is empty, then it is replenished with a call to `read`. Receiving a short count from this invocation of `read` is not an error; it simply has the effect of partially filling the read buffer. Once the buffer is

code/src/csapp.c

```
static ssize_t rio_read(rio_t *rp, char *usrbuf, size_t n)
{
    int cnt;

    while (rp->rio_cnt <= 0) {  /* Refill if buf is empty */
        rp->rio_cnt = read(rp->rio_fd, rp->rio_buf,
                           sizeof(rp->rio_buf));
        if (rp->rio_cnt < 0) {
            if (errno != EINTR) /* Interrupted by sig handler return */
                return -1;
        }
        else if (rp->rio_cnt == 0)  /* EOF */
            return 0;
        else
            rp->rio_bufptr = rp->rio_buf; /* Reset buffer ptr */
    }

    /* Copy min(n, rp->rio_cnt) bytes from internal buf to user buf */
    cnt = n;
    if (rp->rio_cnt < n)
        cnt = rp->rio_cnt;
    memcpy(usrbuf, rp->rio_bufptr, cnt);
    rp->rio_bufptr += cnt;
    rp->rio_cnt -= cnt;
    return cnt;
}
```

code/src/csapp.c

Figure 10.7 The internal `rio_read` function.

nonempty, `rio_read` copies the minimum of `n` and `rp->rio_cnt` bytes from the read buffer to the user buffer and returns the number of bytes copied.

To an application program, the `rio_read` function has the same semantics as the Linux `read` function. On error, it returns −1 and sets `errno` appropriately. On EOF, it returns 0. It returns a short count if the number of requested bytes exceeds the number of unread bytes in the read buffer. The similarity of the two functions makes it easy to build different kinds of buffered read functions by substituting `rio_read` for `read`. For example, the `rio_readnb` function in Figure 10.8 has the same structure as `rio_readn`, with `rio_read` substituted for `read`. Similarly, the `rio_readlineb` routine in Figure 10.8 calls `rio_read` at most `maxlen-1` times. Each call returns 1 byte from the read buffer, which is then checked for being the terminating newline.

code/src/csapp.c

```
ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen)
{
    int n, rc;
    char c, *bufp = usrbuf;

    for (n = 1; n < maxlen; n++) {
        if ((rc = rio_read(rp, &c, 1)) == 1) {
            *bufp++ = c;
            if (c == '\n') {
                n++;
                break;
            }
        } else if (rc == 0) {
            if (n == 1)
                return 0; /* EOF, no data read */
            else
                break;    /* EOF, some data was read */
        } else
            return -1;    /* Error */
    }
    *bufp = 0;
    return n-1;
}
```

code/src/csapp.c

code/src/csapp.c

```
ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n)
{
    size_t nleft = n;
    ssize_t nread;
    char *bufp = usrbuf;

    while (nleft > 0) {
        if ((nread = rio_read(rp, bufp, nleft)) < 0)
            return -1;          /* errno set by read() */
        else if (nread == 0)
            break;              /* EOF */
        nleft -= nread;
        bufp += nread;
    }
    return (n - nleft);         /* Return >= 0 */
}
```

code/src/csapp.c

Figure 10.8 **The `rio_readlineb` and `rio_readnb` functions.**

Aside Origins of the Rio package

The Rio functions are inspired by the `readline`, `readn`, and `writen` functions described by W. Richard Stevens in his classic network programming text [110]. The `rio_readn` and `rio_writen` functions are identical to the Stevens `readn` and `writen` functions. However, the Stevens `readline` function has some limitations that are corrected in Rio. First, because `readline` is buffered and `readn` is not, these two functions cannot be used together on the same descriptor. Second, because it uses a `static` buffer, the Stevens `readline` function is not thread-safe, which required Stevens to introduce a different thread-safe version called `readline_r`. We have corrected both of these flaws with the `rio_readlineb` and `rio_readnb` functions, which are mutually compatible and thread-safe.

10.6 Reading File Metadata

An application can retrieve information about a file (sometimes called the file's *metadata*) by calling the `stat` and `fstat` functions.

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

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

Returns: 0 if OK, −1 on error

The `stat` function takes as input a filename and fills in the members of a `stat` structure shown in Figure 10.9. The `fstat` function is similar, but it takes a file descriptor instead of a filename. We will need the `st_mode` and `st_size` members of the `stat` structure when we discuss Web servers in Section 11.5. The other members are beyond our scope.

The `st_size` member contains the file size in bytes. The `st_mode` member encodes both the file permission bits (Figure 10.2) and the file type (Section 10.2). Linux defines macro predicates in `sys/stat.h` for determining the file type from the `st_mode` member:

S_ISREG(m). Is this a regular file?

S_ISDIR(m). Is this a directory file?

S_ISSOCK(m). Is this a network socket?

Figure 10.10 shows how we might use these macros and the `stat` function to read and interpret a file's `st_mode` bits.

statbuf.h (included by sys/stat.h)

```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t         st_dev;      /* Device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* Protection and file type */
    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 type (if inode device) */
    off_t         st_size;     /* Total size, in bytes */
    unsigned long st_blksize;  /* Block size for filesystem I/O */
    unsigned long st_blocks;   /* Number of blocks allocated */
    time_t        st_atime;    /* Time of last access */
    time_t        st_mtime;    /* Time of last modification */
    time_t        st_ctime;    /* Time of last change */
};
```

statbuf.h (included by sys/stat.h)

Figure 10.9 **The `stat` structure.**

code/io/statcheck.c

```
#include "csapp.h"

int main (int argc, char **argv)
{
    struct stat stat;
    char *type, *readok;

    Stat(argv[1], &stat);
    if (S_ISREG(stat.st_mode))     /* Determine file type */
        type = "regular";
    else if (S_ISDIR(stat.st_mode))
        type = "directory";
    else
        type = "other";
    if ((stat.st_mode & S_IRUSR)) /* Check read access */
        readok = "yes";
    else
        readok = "no";

    printf("type: %s, read: %s\n", type, readok);
    exit(0);
}
```

code/io/statcheck.c

Figure 10.10 **Querying and manipulating a file's `st_mode` bits.**

10.7 Reading Directory Contents

Applications can read the contents of a directory with the `readdir` family of functions.

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

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

Returns: pointer to handle if OK, NULL on error

The `opendir` function takes a pathname and returns a pointer to a *directory stream*. A stream is an abstraction for an ordered list of items, in this case a list of directory entries.

```
#include <dirent.h>

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

Returns: pointer to next directory entry if OK, NULL if no more entries or error

Each call to `readdir` returns a pointer to the next directory entry in the stream `dirp`, or NULL if there are no more entries. Each directory entry is a structure of the form

```
struct dirent {
    ino_t d_ino;        /* inode number */
    char  d_name[256]; /* Filename */
};
```

Although some versions of Linux include other structure members, these are the only two that are standard across all systems. The `d_name` member is the filename, and `d_ino` is the file location.

On error, `readdir` returns NULL and sets `errno`. Unfortunately, the only way to distinguish an error from the end-of-stream condition is to check if `errno` has been modified since the call to `readdir`.

```
#include <dirent.h>

int closedir(DIR *dirp);
```

Returns: 0 on success, −1 on error

The `closedir` function closes the stream and frees up any of its resources. Figure 10.11 shows how we might use `readdir` to read the contents of a directory.

code/io/readdir.c

```
#include "csapp.h"

int main(int argc, char **argv)
{
    DIR *streamp;
    struct dirent *dep;

    streamp = Opendir(argv[1]);

    errno = 0;
    while ((dep = readdir(streamp)) != NULL) {
        printf("Found file: %s\n", dep->d_name);
    }
    if (errno != 0)
        unix_error("readdir error");

    Closedir(streamp);
    exit(0);
}
```

code/io/readdir.c

Figure 10.11 Reading the contents of a directory.

10.8 Sharing Files

Linux files can be shared in a number of different ways. Unless you have a clear picture of how the kernel represents open files, the idea of file sharing can be quite confusing. The kernel represents open files using three related data structures:

Descriptor table. Each process has its own separate *descriptor table* whose entries are indexed by the process's open file descriptors. Each open descriptor entry points to an entry in the *file table.*

File table. The set of open files is represented by a file table that is shared by all processes. Each file table entry consists of (for our purposes) the current file position, a *reference count* of the number of descriptor entries that currently point to it, and a pointer to an entry in the *v-node table*. Closing a descriptor decrements the reference count in the associated file table entry. The kernel will not delete the file table entry until its reference count is zero.

v-node table. Like the file table, the v-node table is shared by all processes. Each entry contains most of the information in the `stat` structure, including the `st_mode` and `st_size` members.

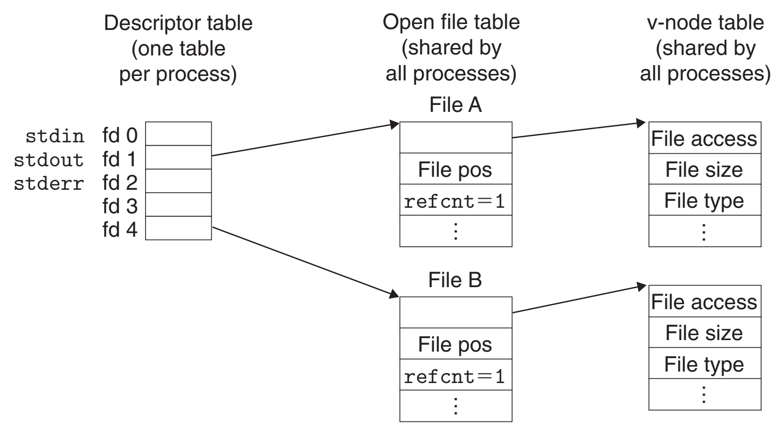

Figure 10.12
Typical kernel data structures for open files. In this example, two descriptors reference distinct files. There is no sharing.

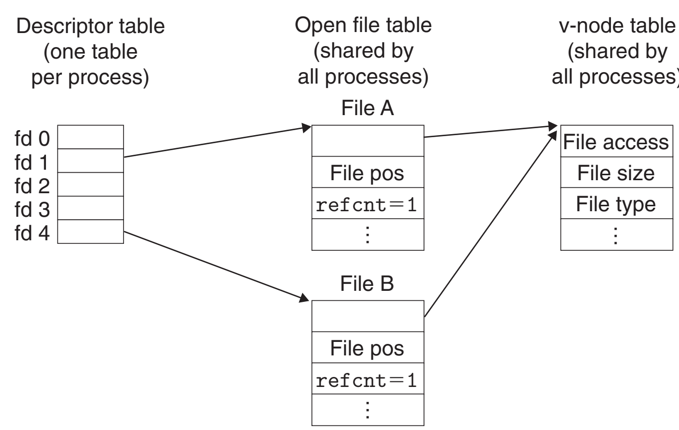

Figure 10.13
File sharing. This example shows two descriptors sharing the same disk file through two open file table entries.

Figure 10.12 shows an example where descriptors 1 and 4 reference two different files through distinct open file table entries. This is the typical situation, where files are not shared and where each descriptor corresponds to a distinct file.

Multiple descriptors can also reference the same file through different file table entries, as shown in Figure 10.13. This might happen, for example, if you were to call the `open` function twice with the same filename. The key idea is that each descriptor has its own distinct file position, so different reads on different descriptors can fetch data from different locations in the file.

We can also understand how parent and child processes share files. Suppose that before a call to `fork`, the parent process has the open files shown in Figure 10.12. Then Figure 10.14 shows the situation after the call to `fork`.

The child gets its own duplicate copy of the parent's descriptor table. Parent and child share the same set of open file tables and thus share the same file position. An important consequence is that the parent and child must both close their descriptors before the kernel will delete the corresponding file table entry.

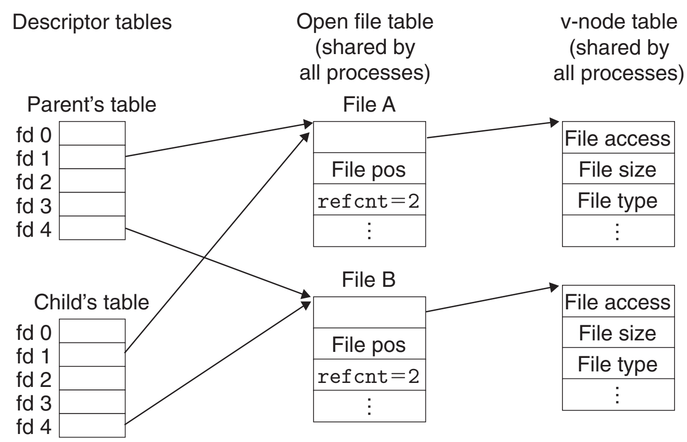

Figure 10.14 **How a child process inherits the parent's open files.** The initial situation is in Figure 10.12.

Practice Problem 10.2 (solution page 951)

Suppose the disk file `foobar.txt` consists of the six ASCII characters `foobar`. Then what is the output of the following program?

```
#include "csapp.h"

int main()
{
    int fd1, fd2;
    char c;

    fd1 = Open("foobar.txt", O_RDONLY, 0);
    fd2 = Open("foobar.txt", O_RDONLY, 0);
    Read(fd1, &c, 1);
    Read(fd2, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```

Practice Problem 10.3 (solution page 951)

As before, suppose the disk file `foobar.txt` consists of the six ASCII characters `foobar`. Then what is the output of the following program?

```
#include "csapp.h"

int main()
{
    int fd;
    char c;
```

```

    fd = Open("foobar.txt", O_RDONLY, 0);
    if (Fork() == 0) {
        Read(fd, &c, 1);
        exit(0);
    }
    Wait(NULL);
    Read(fd, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```

10.9 I/O Redirection

Linux shells provide *I/O redirection* operators that allow users to associate standard input and output with disk files. For example, typing

```
linux> ls > foo.txt
```

causes the shell to load and execute the `ls` program, with standard output redirected to disk file `foo.txt`. As we will see in Section 11.5, a Web server performs a similar kind of redirection when it runs a CGI program on behalf of the client. So how does I/O redirection work? One way is to use the `dup2` function.

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```

Returns: nonnegative descriptor if OK, −1 on error

The `dup2` function copies descriptor table entry `oldfd` to descriptor table entry `newfd`, overwriting the previous contents of descriptor table entry `newfd`. If `newfd` was already open, then `dup2` closes `newfd` before it copies `oldfd`.

Suppose that before calling `dup2(4,1)`, we have the situation in Figure 10.12, where descriptor 1 (standard output) corresponds to file A (say, a terminal) and descriptor 4 corresponds to file B (say, a disk file). The reference counts for A and B are both equal to 1. Figure 10.15 shows the situation after calling `dup2(4,1)`. Both descriptors now point to file B; file A has been closed and its file table and v-node table entries deleted; and the reference count for file B has been incremented. From this point on, any data written to standard output are redirected to file B.

Practice Problem 10.4 (solution page 951)

How would you use `dup2` to redirect standard input to descriptor 5?

Aside Right and left hoinkies

To avoid confusion with other bracket-type operators such as ‘]’ and ‘[’, we have always referred to the shell’s ‘>’ operator as a “right hoinky” and the ‘<’ operator as a “left hoinky.”

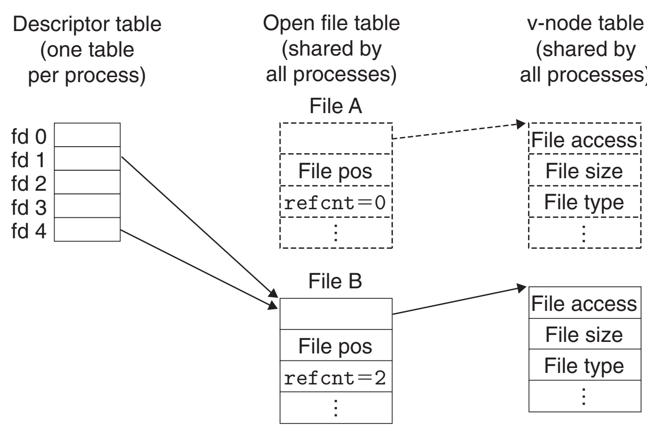

Figure 10.15 **Kernel data structures after redirecting standard output by calling** `dup2(4,1)`. The initial situation is shown in Figure 10.12.

Practice Problem 10.5 (solution page 952)

Assuming that the disk file `foobar.txt` consists of the six ASCII characters `foobar`, what is the output of the following program?

```
#include "csapp.h"

int main()
{
    int fd1, fd2;
    char c;

    fd1 = Open("foobar.txt", O_RDONLY, 0);
    fd2 = Open("foobar.txt", O_RDONLY, 0);
    Read(fd2, &c, 1);
    Dup2(fd2, fd1);
    Read(fd1, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```

10.10 Standard I/O

The C language defines a set of higher-level input and output functions, called the *standard I/O library*, that provides programmers with a higher-level alternative to Unix I/O. The library (`libc`) provides functions for opening and closing files (`fopen` and `fclose`), reading and writing bytes (`fread` and `fwrite`), reading and writing strings (`fgets` and `fputs`), and sophisticated formatted I/O (`scanf` and `printf`).

The standard I/O library models an open file as a *stream*. To the programmer, a stream is a pointer to a structure of type `FILE`. Every ANSI C program begins with three open streams, `stdin`, `stdout`, and `stderr`, which correspond to standard input, standard output, and standard error, respectively:

```
#include <stdio.h>
extern FILE *stdin;    /* Standard input (descriptor 0) */
extern FILE *stdout;   /* Standard output (descriptor 1) */
extern FILE *stderr;   /* Standard error (descriptor 2) */
```

A stream of type FILE is an abstraction for a file descriptor and a *stream buffer*. The purpose of the stream buffer is the same as the Rio read buffer: to minimize the number of expensive Linux I/O system calls. For example, suppose we have a program that makes repeated calls to the standard I/O `getc` function, where each invocation returns the next character from a file. When `getc` is called the first time, the library fills the stream buffer with a single call to the `read` function and then returns the first byte in the buffer to the application. As long as there are unread bytes in the buffer, subsequent calls to `getc` can be served directly from the stream buffer.

10.11 Putting It Together: Which I/O Functions Should I Use?

Figure 10.16 summarizes the various I/O packages that we have discussed in this chapter.

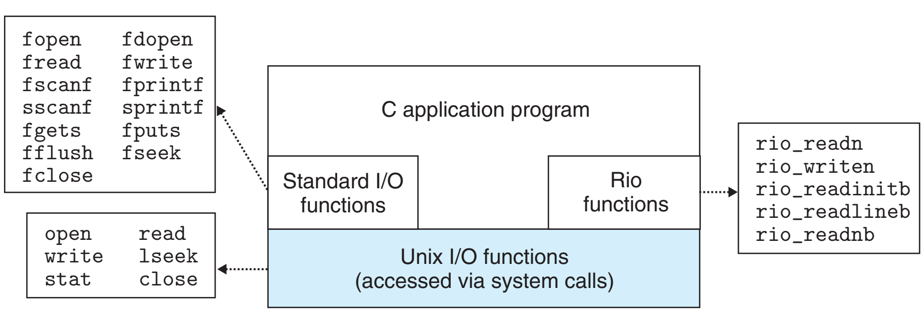

Figure 10.16 Relationship between Unix I/O, standard I/O, and Rio.

The Unix I/O model is implemented in the operating system kernel. It is available to applications through functions such as `open`, `close`, `lseek`, `read`, `write`, and `stat`. The higher-level Rio and standard I/O functions are implemented "on top of" (using) the Unix I/O functions. The Rio functions are robust wrappers for `read` and `write` that were developed specifically for this textbook. They automatically deal with short counts and provide an efficient buffered approach for reading text lines. The standard I/O functions provide a more complete buffered alternative to the Unix I/O functions, including formatted I/O routines such as `printf` and `scanf`.

So which of these functions should you use in your programs? Here are some basic guidelines:

G1: *Use the standard I/O functions whenever possible.* The standard I/O functions are the method of choice for I/O on disk and terminal devices. Most C programmers use standard I/O exclusively throughout their careers, never bothering with the lower-level Unix I/O functions (except possibly `stat`, which has no counterpart in the standard I/O library). Whenever possible, we recommend that you do likewise.

G2: *Don't use* `scanf` *or* `rio_readlineb` *to read binary files.* Functions like `scanf` and `rio_readlineb` are designed specifically for reading text files. A common error that students make is to use these functions to read binary data, causing their programs to fail in strange and unpredictable ways. For example, binary files might be littered with many `0xa` bytes that have nothing to do with terminating text lines.

G3: *Use the* Rio *functions for I/O on network sockets.* Unfortunately, standard I/O poses some nasty problems when we attempt to use it for input and output on networks. As we will see in Section 11.4, the Linux abstraction for a network is a type of file called a *socket*. Like any Linux file, sockets are referenced by file descriptors, known in this case as *socket descriptors*. Application processes communicate with processes running on other computers by reading and writing socket descriptors.

Standard I/O streams are *full duplex* in the sense that programs can perform input and output on the same stream. However, there are poorly documented restrictions on streams that interact badly with restrictions on sockets:

Restriction 1: *Input functions following output functions.* An input function cannot follow an output function without an intervening call to `fflush`, `fseek`, `fsetpos`, or `rewind`. The `fflush` function empties the buffer associated with a stream. The latter three functions use the Unix I/O `lseek` function to reset the current file position.

Restriction 2: *Output functions following input functions.* An output function cannot follow an input function without an intervening call to `fseek`, `fsetpos`, or `rewind`, unless the input function encounters an end-of-file.

These restrictions pose a problem for network applications because it is illegal to use the `lseek` function on a socket. The first restriction on stream I/O can be worked around by adopting a discipline of flushing the buffer before every input operation. However, the only way to work around the second restriction is to open two streams on the same open socket descriptor, one for reading and one for writing:

```
FILE *fpin, *fpout;

fpin = fdopen(sockfd, "r");
fpout = fdopen(sockfd, "w");
```

But this approach has problems as well, because it requires the application to call `fclose` on both streams in order to free the memory resources associated with each stream and avoid a memory leak:

```
fclose(fpin);
fclose(fpout);
```

Each of these operations attempts to close the same underlying socket descriptor, so the second `close` operation will fail. This is not a problem for sequential programs, but closing an already closed descriptor in a threaded program is a recipe for disaster (see Section 12.7.4).

Thus, we recommend that you not use the standard I/O functions for input and output on network sockets. Use the robust Rio functions instead. If you need formatted output, use the `sprintf` function to format a string in memory, and then send it to the socket using `rio_writen`. If you need formatted input, use `rio_readlineb` to read an entire text line, and then use `sscanf` to extract different fields from the text line.

10.12 Summary

Linux provides a small number of system-level functions, based on the Unix I/O model, that allow applications to open, close, read, and write files, to fetch file metadata, and to perform I/O redirection. Linux read and write operations are subject to short counts that applications must anticipate and handle correctly. Instead of calling the Unix I/O functions directly, applications should use the Rio package, which deals with short counts automatically by repeatedly performing read and write operations until all of the requested data have been transferred.

The Linux kernel uses three related data structures to represent open files. Entries in a descriptor table point to entries in the open file table, which point to entries in the v-node table. Each process has its own distinct descriptor table, while all processes share the same open file and v-node tables. Understanding the general organization of these structures clarifies our understanding of both file sharing and I/O redirection.

The standard I/O library is implemented on top of Unix I/O and provides a powerful set of higher-level I/O routines. For most applications, standard I/O is the simpler, preferred alternative to Unix I/O. However, because of some mutually incompatible restrictions on standard I/O and network files, Unix I/O, rather than standard I/O, should be used for network applications.

Bibliographic Notes

Kerrisk gives a comprehensive treatment of Unix I/O and the Linux file system [62]. Stevens wrote the original standard reference text for Unix I/O [111]. Kernighan and Ritchie give a clear and complete discussion of the standard I/O functions [61].

Homework Problems

10.6 ◆
What is the output of the following program?

```
#include "csapp.h"

int main()
{
    int fd1, fd2;

    fd1 = Open("foo.txt", O_RDONLY, 0);
    fd2 = Open("bar.txt", O_RDONLY, 0);
    Close(fd2);
    fd2 = Open("baz.txt", O_RDONLY, 0);
    printf("fd2 = %d\n", fd2);
    exit(0);
}
```

10.7 ◆
Modify the `cpfile` program in Figure 10.5 so that it uses the Rio functions to copy standard input to standard output, MAXBUF bytes at a time.

10.8 ◆◆
Write a version of the `statcheck` program in Figure 10.10, called `fstatcheck`, that takes a descriptor number on the command line rather than a filename.

10.9 ◆◆
Consider the following invocation of the `fstatcheck` program from Problem 10.8:

```
linux> fstatcheck 3 < foo.txt
```

You might expect that this invocation of `fstatcheck` would fetch and display metadata for file `foo.txt`. However, when we run it on our system, it fails with a “bad file descriptor.” Given this behavior, fill in the pseudocode that the shell must be executing between the `fork` and `execve` calls:

```
if (Fork() == 0) { /* child */
    /* What code is the shell executing right here? */
    Execve("fstatcheck", argv, envp);
}
```

10.10 ◆◆

Modify the `cpfile` program in Figure 10.5 so that it takes an optional command-line argument `infile`. If `infile` is given, then copy `infile` to standard output; otherwise, copy standard input to standard output as before. The twist is that your solution must use the original copy loop (lines 9–11) for both cases. You are only allowed to insert code, and you are not allowed to change any of the existing code.

Solutions to Practice Problems

Solution to Problem 10.1 (page 931)

Unix processes begin life with open descriptors assigned to `stdin` (descriptor 0), `stdout` (descriptor 1), and `stderr` (descriptor 2). The `open` function always returns the lowest unopened descriptor, so the first call to `open` returns descriptor 3. The call to the `close` function frees up descriptor 3. The final call to `open` returns descriptor 3, and thus the output of the program is `fd2 = 3`.

Solution to Problem 10.2 (page 944)

The descriptors `fd1` and `fd2` each have their own open file table entry, so each descriptor has its own file position for `foobar.txt`. Thus, the read from `fd2` reads the first byte of `foobar.txt`, and the output is

```
c = f
```

and not

```
c = o
```

as you might have thought initially.

Solution to Problem 10.3 (page 944)

Recall that the child inherits the parent's descriptor table and that all processes shared the same open file table. Thus, the descriptor `fd` in both the parent and child points to the same open file table entry. When the child reads the first byte of the file, the file position increases by 1. Thus, the parent reads the second byte, and the output is

```
c = o
```

Solution to Problem 10.4 (page 945)

To redirect standard input (descriptor 0) to descriptor 5, we would call `dup2(5,0)`, or equivalently, `dup2(5,STDIN_FILENO)`.

Solution to Problem 10.5 (page 946)

At first glance, you might think the output would be

```
c = f
```

but because we are redirecting `fd1` to `fd2`, the output is really

```
c = o
```