# 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 Unix 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 dependences 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 Unix *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 Unix 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 Unix 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 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 or'd 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.

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 10.1. 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:

| 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 10.1: **Access permission bits.** Defined in `sys/stat.h`.

```
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: zero if OK, −1 on error

Closing a descriptor that is already closed is an error.

**Practice Problem 10.1**:

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);
}
```

## 10.3 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 10.2 shows a program that uses `read` and `write` calls to copy the standard input to the standard output, 1 byte at a time.

*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 10.2: **Copies standard input to standard output one byte at a time.**

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

**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? A `size_t` is defined as an `unsigned int`, and an `ssize_t` (*signed size*) is defined as an `int`. 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 two, from 4 GB down to 2 GB. **End Aside.**

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:

- *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 zero.

- *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 11.3.3), 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 Unix *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.

## 10.4 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 [109], 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.

### 10.4.1 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 10.3 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. (See Section 8.5.4 for a discussion on interrupted system calls).

### 10.4.2 RIO Buffered Input Functions

A *text line* is a sequence of ASCII characters terminated by a newline character. On Unix systems, the newline character (‘`\n`’) is the same as the ASCII line feed character (LF) and has a numeric value of `0x0a`. Suppose we wanted to write a program that counts the number of text 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);
```

Return: 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.

*code/src/csapp.c*

```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*

```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 10.3: **The `rio_readn` and `rio_writen` functions.**

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 10.4 shows how to use the RIO functions to copy a text file from standard input to standard output, one line at a time.

*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 10.4: **Copying a text file from standard input to standard output.**

Figure 10.5 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.

The heart of the RIO read routines is the `rio_read` function shown in Figure 10.6. The `rio_read` function is a buffered version of the Unix `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, and simply has the effect of partially filling the read buffer. Once the buffer is 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 Unix `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 10.7 has the same structure as `rio_readn`, with `rio_read` substituted for `read`. Similarly, the `rio_readlineb` routine in Figure 10.7 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.

**Aside: Origins of the RIO package.**

*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 10.5: **A read buffer of type `rio_t` and the `rio_readinitb` function that initializes it.**

The RIO functions are inspired by the `readline`, `readn`, and `writen` functions described by W. Richard Stevens in his classic network programming text [109]. 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. **End Aside.**

## 10.5 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 file name and fills in the members of a `stat` structure shown in

*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 10.6: **The internal `rio_read` function.**

*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')
                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;
}
```

*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) {
            if (errno == EINTR) /* interrupted by sig handler return */
                nread = 0;      /* 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*

Figure 10.7: **The `rio_readlineb` and `rio_readnb` functions.**

Figure 10.8. The `fstat` function is similar, but takes a file descriptor instead of a file name. 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.

*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;  /* blocksize 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 10.8: **The `stat` structure.**

The `st_size` member contains the file size in bytes. The `st_mode` member encodes both the file permission bits (Figure 10.1) and the *file type*. Unix recognizes a number of different file types. A *regular file* contains some sort of binary or text data. To the kernel there is no difference between text files and binary files. A *directory file* contains information about other files. A *socket* is a file that is used to communicate with another process across a network (Section 11.4).

Unix provides macro predicates for determining the file type from the `st_mode` member. Figure 10.9 lists a subset of these macros.

| Macro | Description |
|---|---|
| S_ISREG() | Is this a regular file? |
| S_ISDIR() | Is this a directory file? |
| S_ISSOCK() | Is this a network socket? |

Figure 10.9: **Macros for determining file type from the `st_mode` bits.** Defined in `sys/stat.h`

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

*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.**

## 10.6 Sharing Files

Unix 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 10.11 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.

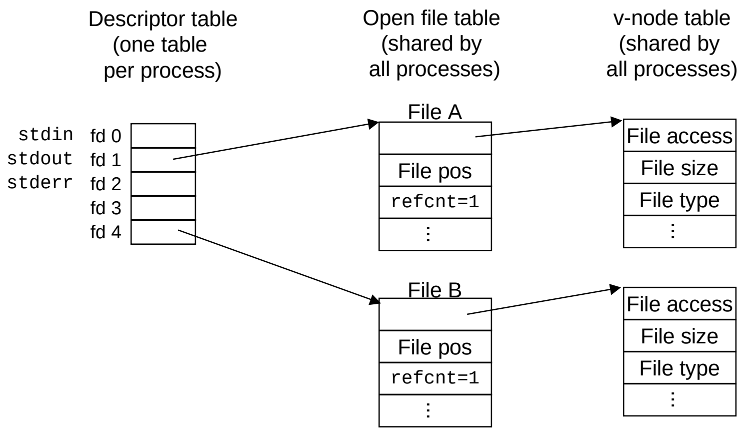

Figure 10.11: **Typical kernel data structures for open files.** In this example, two descriptors reference distinct files. There is no sharing.

Multiple descriptors can also reference the same file through different file table entries, as shown in Figure 10.12. 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 10.11. Then Figure 10.13 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