3

File I/O

3.1 Introduction

We'll start our discussion of the UNIX System by describing the functions available for file I/O—open a file, read a file, write a file, and so on. Most file I/O on a UNIX system can be performed using only five functions: `open`, `read`, `write`, `lseek`, and `close`. We then examine the effect of various buffer sizes on the `read` and `write` functions.

The functions described in this chapter are often referred to as *unbuffered I/O*, in contrast to the standard I/O routines, which we describe in Chapter 5. The term *unbuffered* means that each `read` or `write` invokes a system call in the kernel. These unbuffered I/O functions are not part of ISO C, but are part of POSIX.1 and the Single UNIX Specification.

Whenever we describe the sharing of resources among multiple processes, the concept of an atomic operation becomes important. We examine this concept with regard to file I/O and the arguments to the `open` function. This leads to a discussion of how files are shared among multiple processes and which kernel data structures are involved. After describing these features, we describe the `dup`, `fcntl`, `sync`, `fsync`, and `ioctl` functions.

3.2 File Descriptors

To the kernel, all open files are referred to by file descriptors. A file descriptor is a non-negative integer. When we open an existing file or create a new file, the kernel returns a file descriptor to the process. When we want to read or write a file, we identify the file with the file descriptor that was returned by `open` or `creat` as an argument to either `read` or `write`.

By convention, UNIX System shells associate file descriptor 0 with the standard input of a process, file descriptor 1 with the standard output, and file descriptor 2 with the standard error. This convention is used by the shells and many applications; it is not a feature of the UNIX kernel. Nevertheless, many applications would break if these associations weren't followed.

Although their values are standardized by POSIX.1, the magic numbers 0, 1, and 2 should be replaced in POSIX-compliant applications with the symbolic constants `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO` to improve readability. These constants are defined in the `<unistd.h>` header.

File descriptors range from 0 through `OPEN_MAX-1`. (Recall Figure 2.11.) Early historical implementations of the UNIX System had an upper limit of 19, allowing a maximum of 20 open files per process, but many systems subsequently increased this limit to 63.

> With FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10, the limit is essentially infinite, bounded by the amount of memory on the system, the size of an integer, and any hard and soft limits configured by the system administrator.

3.3 `open` and `openat` Functions

A file is opened or created by calling either the `open` function or the `openat` function.

```
#include <fcntl.h>

int open(const char *path, int oflag, ... /* mode_t mode */ );

int openat(int fd, const char *path, int oflag, ... /* mode_t mode */ );
```

Both return: file descriptor if OK, –1 on error

We show the last argument as `...`, which is the ISO C way to specify that the number and types of the remaining arguments may vary. For these functions, the last argument is used only when a new file is being created, as we describe later. We show this argument as a comment in the prototype.

The *path* parameter is the name of the file to open or create. This function has a multitude of options, which are specified by the *oflag* argument. This argument is formed by ORing together one or more of the following constants from the `<fcntl.h>` header:

`O_RDONLY` Open for reading only.

`O_WRONLY` Open for writing only.

`O_RDWR` Open for reading and writing.

> Most implementations define `O_RDONLY` as 0, `O_WRONLY` as 1, and `O_RDWR` as 2, for compatibility with older programs.

`O_EXEC` Open for execute only.

`O_SEARCH` Open for search only (applies to directories).

> The purpose of the O_SEARCH constant is to evaluate search permissions at the time a directory is opened. Further operations using the directory's file descriptor will not reevaluate permission to search the directory. None of the versions of the operating systems covered in this book support O_SEARCH yet.

One and only one of the previous five constants must be specified. The following constants are optional:

O_APPEND Append to the end of file on each write. We describe this option in detail in Section 3.11.

O_CLOEXEC Set the FD_CLOEXEC file descriptor flag. We discuss file descriptor flags in Section 3.14.

O_CREAT Create the file if it doesn't exist. This option requires a third argument to the open function (a fourth argument to the openat function)—the *mode*, which specifies the access permission bits of the new file. (When we describe a file's access permission bits in Section 4.5, we'll see how to specify the *mode* and how it can be modified by the umask value of a process.)

O_DIRECTORY Generate an error if *path* doesn't refer to a directory.

O_EXCL Generate an error if O_CREAT is also specified and the file already exists. This test for whether the file already exists and the creation of the file if it doesn't exist is an atomic operation. We describe atomic operations in more detail in Section 3.11.

O_NOCTTY If *path* refers to a terminal device, do not allocate the device as the controlling terminal for this process. We talk about controlling terminals in Section 9.6.

O_NOFOLLOW Generate an error if *path* refers to a symbolic link. We discuss symbolic links in Section 4.17.

O_NONBLOCK If *path* refers to a FIFO, a block special file, or a character special file, this option sets the nonblocking mode for both the opening of the file and subsequent I/O. We describe this mode in Section 14.2.

> In earlier releases of System V, the O_NDELAY (no delay) flag was introduced. This option is similar to the O_NONBLOCK (nonblocking) option, but an ambiguity was introduced in the return value from a read operation. The no-delay option causes a read operation to return 0 if there is no data to be read from a pipe, FIFO, or device, but this conflicts with a return value of 0, indicating an end of file. SVR4-based systems still support the no-delay option, with the old semantics, but new applications should use the nonblocking option instead.

O_SYNC Have each write wait for physical I/O to complete, including I/O necessary to update file attributes modified as a result of the write. We use this option in Section 3.14.

O_TRUNC If the file exists and if it is successfully opened for either write-only or read–write, truncate its length to 0.

`O_TTY_INIT` When opening a terminal device that is not already open, set the nonstandard `termios` parameters to values that result in behavior that conforms to the Single UNIX Specification. We discuss the `termios` structure when we discuss terminal I/O in Chapter 18.

The following two flags are also optional. They are part of the synchronized input and output option of the Single UNIX Specification (and thus POSIX.1).

`O_DSYNC` Have each `write` wait for physical I/O to complete, but don't wait for file attributes to be updated if they don't affect the ability to read the data just written.

> The `O_DSYNC` and `O_SYNC` flags are similar, but subtly different. The `O_DSYNC` flag affects a file's attributes only when they need to be updated to reflect a change in the file's data (for example, update the file's size to reflect more data). With the `O_SYNC` flag, data and attributes are always updated synchronously. When overwriting an existing part of a file opened with the `O_DSYNC` flag, the file times wouldn't be updated synchronously. In contrast, if we had opened the file with the `O_SYNC` flag, every `write` to the file would update the file's times before the `write` returns, regardless of whether we were writing over existing bytes or appending to the file.

`O_RSYNC` Have each `read` operation on the file descriptor wait until any pending writes for the same portion of the file are complete.

> Solaris 10 supports all three synchronization flags. Historically, FreeBSD (and thus Mac OS X) have used the `O_FSYNC` flag, which has the same behavior as `O_SYNC`. Because the two flags are equivalent, they define the flags to have the same value. FreeBSD 8.0 doesn't support the `O_DSYNC` or `O_RSYNC` flags. Mac OS X doesn't support the `O_RSYNC` flag, but defines the `O_DSYNC` flag, treating it the same as the `O_SYNC` flag. Linux 3.2.0 supports the `O_DSYNC` flag, but treats the `O_RSYNC` flag the same as `O_SYNC`.

The file descriptor returned by `open` and `openat` is guaranteed to be the lowest-numbered unused descriptor. This fact is used by some applications to open a new file on standard input, standard output, or standard error. For example, an application might close standard output—normally, file descriptor 1—and then open another file, knowing that it will be opened on file descriptor 1. We'll see a better way to guarantee that a file is open on a given descriptor in Section 3.12, when we explore the `dup2` function.

The *fd* parameter distinguishes the `openat` function from the `open` function. There are three possibilities:

1. The *path* parameter specifies an absolute pathname. In this case, the *fd* parameter is ignored and the `openat` function behaves like the `open` function.
2. The *path* parameter specifies a relative pathname and the *fd* parameter is a file descriptor that specifies the starting location in the file system where the relative pathname is to be evaluated. The *fd* parameter is obtained by opening the directory where the relative pathname is to be evaluated.

3. The *path* parameter specifies a relative pathname and the *fd* parameter has the special value AT_FDCWD. In this case, the pathname is evaluated starting in the current working directory and the openat function behaves like the open function.

The openat function is one of a class of functions added to the latest version of POSIX.1 to address two problems. First, it gives threads a way to use relative pathnames to open files in directories other than the current working directory. As we'll see in Chapter 11, all threads in the same process share the same current working directory, so this makes it difficult for multiple threads in the same process to work in different directories at the same time. Second, it provides a way to avoid time-of-check-to-time-of-use (TOCTTOU) errors.

The basic idea behind TOCTTOU errors is that a program is vulnerable if it makes two file-based function calls where the second call depends on the results of the first call. Because the two calls are not atomic, the file can change between the two calls, thereby invalidating the results of the first call, leading to a program error. TOCTTOU errors in the file system namespace generally deal with attempts to subvert file system permissions by tricking a privileged program into either reducing permissions on a privileged file or modifying a privileged file to open up a security hole. Wei and Pu [2005] discuss TOCTTOU weaknesses in the UNIX file system interface.

Filename and Pathname Truncation

What happens if NAME_MAX is 14 and we try to create a new file in the current directory with a filename containing 15 characters? Traditionally, early releases of System V, such as SVR2, allowed this to happen, silently truncating the filename beyond the 14th character. BSD-derived systems, in contrast, returned an error status, with errno set to ENAMETOOLONG. Silently truncating the filename presents a problem that affects more than simply the creation of new files. If NAME_MAX is 14 and a file exists whose name is exactly 14 characters, any function that accepts a pathname argument, such as open or stat, has no way to determine what the original name of the file was, as the original name might have been truncated.

With POSIX.1, the constant _POSIX_NO_TRUNC determines whether long filenames and long components of pathnames are truncated or an error is returned. As we saw in Chapter 2, this value can vary based on the type of the file system, and we can use fpathconf or pathconf to query a directory to see which behavior is supported.

> Whether an error is returned is largely historical. For example, SVR4-based systems do not generate an error for the traditional System V file system, S5. For the BSD-style file system (known as UFS), however, SVR4-based systems do generate an error. Figure 2.20 illustrates another example: Solaris will return an error for UFS, but not for PCFS, the DOS-compatible file system, as DOS silently truncates filenames that don't fit in an 8.3 format. BSD-derived systems and Linux always return an error.

If _POSIX_NO_TRUNC is in effect, errno is set to ENAMETOOLONG, and an error status is returned if any filename component of the pathname exceeds NAME_MAX.

> Most modern file systems support a maximum of 255 characters for filenames. Because filenames are usually shorter than this limit, this constraint tends to not present problems for most applications.

3.7 read Function

Data is read from an open file with the read function.

```
#include <unistd.h>

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

Returns: number of bytes read, 0 if end of file, –1 on error

If the read is successful, the number of bytes read is returned. If the end of file is encountered, 0 is returned.

There are several cases in which the number of bytes actually read is less than the amount requested:

- When reading from a regular file, if the end of file is reached before the requested number of bytes has been read. For example, if 30 bytes remain until the end of file and we try to read 100 bytes, read returns 30. The next time we call read, it will return 0 (end of file).
- When reading from a terminal device. Normally, up to one line is read at a time. (We'll see how to change this default in Chapter 18.)
- When reading from a network. Buffering within the network may cause less than the requested amount to be returned.
- When reading from a pipe or FIFO. If the pipe contains fewer bytes than requested, read will return only what is available.
- When reading from a record-oriented device. Some record-oriented devices, such as magnetic tape, can return up to a single record at a time.
- When interrupted by a signal and a partial amount of data has already been read. We discuss this further in Section 10.5.

The read operation starts at the file's current offset. Before a successful return, the offset is incremented by the number of bytes actually read.

POSIX.1 changed the prototype for this function in several ways. The classic definition is

```
int read(int fd, char *buf, unsigned nbytes);
```

- First, the second argument was changed from char * to void * to be consistent with ISO C: the type void * is used for generic pointers.
- Next, the return value was required to be a signed integer (ssize_t) to return a positive byte count, 0 (for end of file), or –1 (for an error).
- Finally, the third argument historically has been an unsigned integer, to allow a 16-bit implementation to read or write up to 65,534 bytes at a time. With the 1990 POSIX.1 standard, the primitive system data type ssize_t was introduced to provide the signed return value, and the unsigned size_t was used for the third argument. (Recall the SSIZE_MAX constant from Section 2.5.2.)

3.8 `write` Function

Data is written to an open file with the `write` function.

```
#include <unistd.h>

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

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

The return value is usually equal to the *nbytes* argument; otherwise, an error has occurred. A common cause for a `write` error is either filling up a disk or exceeding the file size limit for a given process (Section 7.11 and Exercise 10.11).

For a regular file, the write operation starts at the file's current offset. If the `O_APPEND` option was specified when the file was opened, the file's offset is set to the current end of file before each write operation. After a successful write, the file's offset is incremented by the number of bytes actually written.

3.9 I/O Efficiency

The program in Figure 3.5 copies a file, using only the `read` and `write` functions.

```
#include "apue.h"

#define BUFFSIZE    4096

int
main(void)
{
    int     n;
    char    buf[BUFFSIZE];

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

Figure 3.5 Copy standard input to standard output

The following caveats apply to this program.

- It reads from standard input and writes to standard output, assuming that these have been set up by the shell before this program is executed. Indeed, all normal UNIX system shells provide a way to open a file for reading on standard input and to create (or rewrite) a file on standard output. This prevents the program from having to open the input and output files, and allows the user to take advantage of the shell's I/O redirection facilities.

Calling pwrite is equivalent to calling lseek followed by a call to write, with similar exceptions.

Creating a File

We saw another example of an atomic operation when we described the O_CREAT and O_EXCL options for the open function. When both of these options are specified, the open will fail if the file already exists. We also said that the check for the existence of the file and the creation of the file was performed as an atomic operation. If we didn't have this atomic operation, we might try

```
if ((fd = open(path, O_WRONLY)) < 0) {
    if (errno == ENOENT) {
        if ((fd = creat(path, mode)) < 0)
            err_sys("creat error");
    } else {
        err_sys("open error");
    }
}
```

The problem occurs if the file is created by another process between the open and the creat. If the file is created by another process between these two function calls, and if that other process writes something to the file, that data is erased when this creat is executed. Combining the test for existence and the creation into a single atomic operation avoids this problem.

In general, the term *atomic operation* refers to an operation that might be composed of multiple steps. If the operation is performed atomically, either all the steps are performed (on success) or none are performed (on failure). It must not be possible for only a subset of the steps to be performed. We'll return to the topic of atomic operations when we describe the link function (Section 4.15) and record locking (Section 14.3).

3.12 dup and dup2 Functions

An existing file descriptor is duplicated by either of the following functions:

```
#include <unistd.h>

int dup(int fd);

int dup2(int fd, int fd2);
```

Both return: new file descriptor if OK, –1 on error

The new file descriptor returned by dup is guaranteed to be the lowest-numbered available file descriptor. With dup2, we specify the value of the new descriptor with the *fd2* argument. If *fd2* is already open, it is first closed. If *fd* equals *fd2*, then dup2 returns *fd2* without closing it. Otherwise, the FD_CLOEXEC file descriptor flag is cleared for *fd2*, so that *fd2* is left open if the process calls exec.

The new file descriptor that is returned as the value of the functions shares the same file table entry as the *fd* argument. We show this in Figure 3.9.

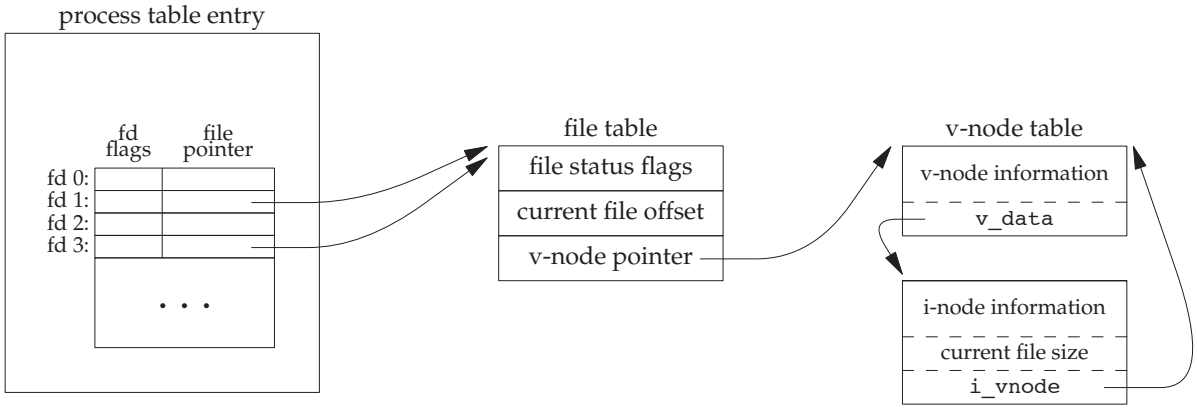

Figure 3.9 Kernel data structures after `dup(1)`

In this figure, we assume that when it's started, the process executes

```
newfd = dup(1);
```

We assume that the next available descriptor is 3 (which it probably is, since 0, 1, and 2 are opened by the shell). Because both descriptors point to the same file table entry, they share the same file status flags—read, write, append, and so on—and the same current file offset.

Each descriptor has its own set of file descriptor flags. As we describe in Section 3.14, the close-on-exec file descriptor flag for the new descriptor is always cleared by the `dup` functions.

Another way to duplicate a descriptor is with the `fcntl` function, which we describe in Section 3.14. Indeed, the call

```
dup(fd);
```

is equivalent to

```
fcntl(fd, F_DUPFD, 0);
```

Similarly, the call

```
dup2(fd, fd2);
```

is equivalent to

```
close(fd2);
fcntl(fd, F_DUPFD, fd2);
```

In this last case, the `dup2` is not exactly the same as a `close` followed by an `fcntl`. The differences are as follows:

15

Interprocess Communication

15.1 Introduction

In Chapter 8, we described the process control primitives and saw how to work with multiple processes. But the only way for these processes to exchange information is by passing open files across a `fork` or an `exec` or through the file system. We'll now describe other techniques for processes to communicate with one another: interprocess communication (IPC).

In the past, UNIX System IPC was a hodgepodge of various approaches, few of which were portable across all UNIX system implementations. Through the POSIX and The Open Group (formerly X/Open) standardization efforts, the situation has since improved, but differences still exist. Figure 15.1 summarizes the various forms of IPC that are supported by the four implementations discussed in this text.

Note that the Single UNIX Specification (the "SUS" column) allows an implementation to support full-duplex pipes, but requires only half-duplex pipes. An implementation that supports full-duplex pipes will still work with correctly written applications that assume that the underlying operating system supports only half-duplex pipes. We use "(full)" instead of a bullet to show implementations that support half-duplex pipes by using full-duplex pipes.

In Figure 15.1, we show a bullet where basic functionality is supported. For full-duplex pipes, if the feature can be provided through UNIX domain sockets (Section 17.2), we show "UDS" in the column. Some implementations support the feature with pipes and UNIX domain sockets, so these entries have both "UDS" and a bullet.

The IPC interfaces introduced as part of the real-time extensions to POSIX.1 were included as options in the Single UNIX Specification. In SUSv4, the semaphore interfaces were moved from an option to the base specification.

IPC type	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
half-duplex pipes	•	(full)	•	•	(full)
FIFOs	•	•	•	•	•
full-duplex pipes	allowed	•, UDS	UDS	UDS	•, UDS
named full-duplex pipes	obsolescent	UDS	UDS	UDS	•, UDS
XSI message queues	XSI	•	•	•	•
XSI semaphores	XSI	•	•	•	•
XSI shared memory	XSI	•	•	•	•
message queues (real-time)	MSG option	•	•		•
semaphores	•	•	•	•	•
shared memory (real-time)	SHM option	•	•	•	•
sockets	•	•	•	•	•
STREAMS	obsolescent				•

Figure 15.1 Summary of UNIX System IPC

Named full-duplex pipes are provided as mounted STREAMS-based pipes, but are marked obsolescent in the Single UNIX Specification.

> Although support for STREAMS on Linux is available in the "Linux Fast-STREAMS" package from the OpenSS7 project, the package hasn't been updated recently. The latest release of the package from 2008 claims to work with kernels up to Linux 2.6.26.

The first ten forms of IPC in Figure 15.1 are usually restricted to IPC between processes on the same host. The final two rows—sockets and STREAMS—are the only two forms that are generally supported for IPC between processes on different hosts.

We have divided the discussion of IPC into three chapters. In this chapter, we examine classical IPC: pipes, FIFOs, message queues, semaphores, and shared memory. In the next chapter, we take a look at network IPC using the sockets mechanism. In Chapter 17, we take a look at some advanced features of IPC.

15.2 Pipes

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have two limitations.

1. Historically, they have been half duplex (i.e., data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.
2. Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls `fork`, and the pipe is used between the parent and the child.

We'll see that FIFOs (Section 15.5) get around the second limitation, and that UNIX domain sockets (Section 17.2) get around both limitations.

Despite these limitations, half-duplex pipes are still the most commonly used form of IPC. Every time you type a sequence of commands in a pipeline for the shell to execute, the shell creates a separate process for each command and links the standard output of one process to the standard input of the next using a pipe.

A pipe is created by calling the `pipe` function.

```
#include <unistd.h>

int pipe(int fd[2]);
```

Returns: 0 if OK, –1 on error

Two file descriptors are returned through the *fd* argument: *fd[0]* is open for reading, and *fd[1]* is open for writing. The output of *fd[1]* is the input for *fd[0]*.

> Originally in 4.3BSD and 4.4BSD, pipes were implemented using UNIX domain sockets. Even though UNIX domain sockets are full duplex by default, these operating systems hobbled the sockets used with pipes so that they operated in half-duplex mode only.
>
> POSIX.1 allows for implementations to support full-duplex pipes. For these implementations, *fd[0]* and *fd[1]* are open for both reading and writing.

Two ways to picture a half-duplex pipe are shown in Figure 15.2. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.

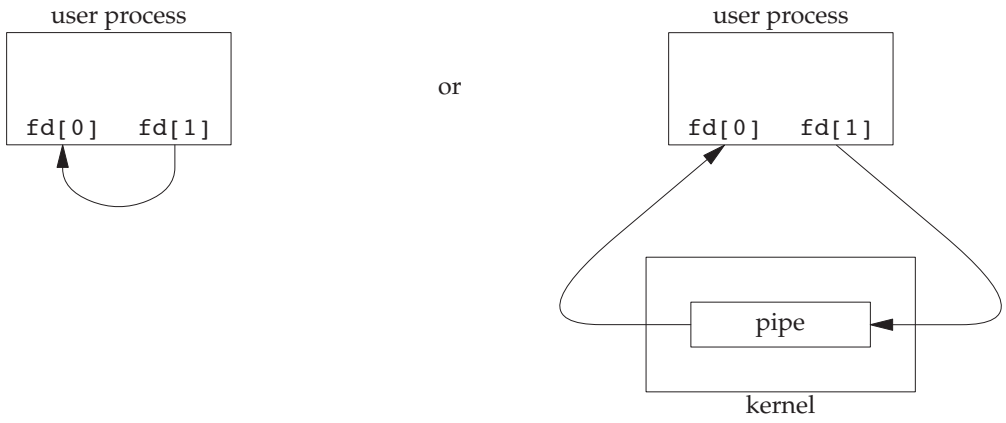

Figure 15.2 Two ways to view a half-duplex pipe

The `fstat` function (Section 4.2) returns a file type of FIFO for the file descriptor of either end of a pipe. We can test for a pipe with the `S_ISFIFO` macro.

> POSIX.1 states that the `st_size` member of the `stat` structure is undefined for pipes. But when the `fstat` function is applied to the file descriptor for the read end of the pipe, many systems store in `st_size` the number of bytes available for reading in the pipe. This is, however, nonportable.

A pipe in a single process is next to useless. Normally, the process that calls `pipe` then calls `fork`, creating an IPC channel from the parent to the child, or vice versa. Figure 15.3 shows this scenario.

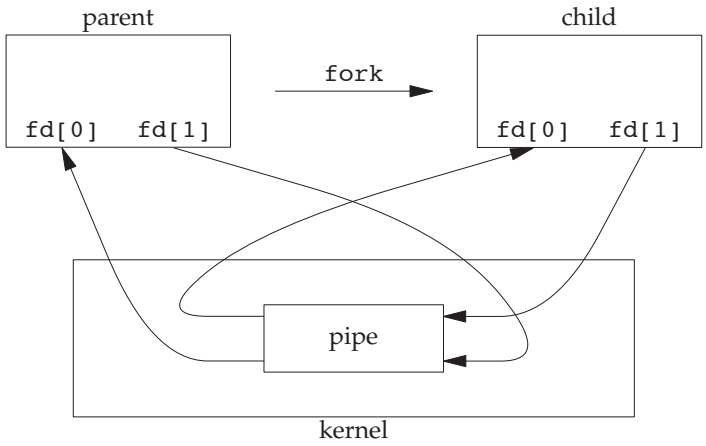

Figure 15.3 Half-duplex pipe after a `fork`

What happens after the `fork` depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe (`fd[0]`), and the child closes the write end (`fd[1]`). Figure 15.4 shows the resulting arrangement of descriptors.

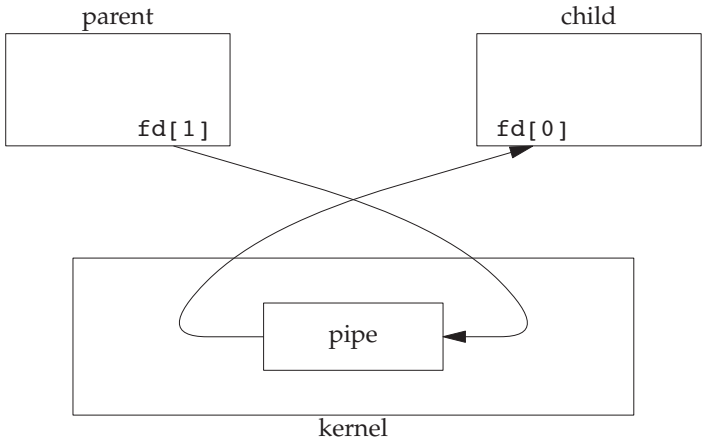

Figure 15.4 Pipe from parent to child

For a pipe from the child to the parent, the parent closes `fd[1]`, and the child closes `fd[0]`.

When one end of a pipe is closed, two rules apply.

1. If we `read` from a pipe whose write end has been closed, `read` returns 0 to indicate an end of file after all the data has been read. (Technically, we should say that this end of file is not generated until there are no more writers for the pipe. It's possible to duplicate a pipe descriptor so that multiple processes have the pipe open for writing. Normally, however, there is a single reader and a single writer for a pipe. When we get to FIFOs in the next section, we'll see that often there are multiple writers for a single FIFO.)

2. If we `write` to a pipe whose read end has been closed, the signal `SIGPIPE` is generated. If we either ignore the signal or catch it and return from the signal handler, `write` returns –1 with `errno` set to `EPIPE`.

When we're writing to a pipe (or FIFO), the constant `PIPE_BUF` specifies the kernel's pipe buffer size. A `write` of `PIPE_BUF` bytes or less will not be interleaved with the `write`s from other processes to the same pipe (or FIFO). But if multiple processes are writing to a pipe (or FIFO), and if we `write` more than `PIPE_BUF` bytes, the data might be interleaved with the data from the other writers. We can determine the value of `PIPE_BUF` by using `pathconf` or `fpathconf` (recall Figure 2.12).

Example

Figure 15.5 shows the code to create a pipe between a parent and its child and to send data down the pipe.

```
#include "apue.h"

int
main(void)
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {       /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                    /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

Figure 15.5 Send data from parent to child over a pipe

Note that the pipe direction here matches the orientation shown in Figure 15.4. □

In the previous example, we called `read` and `write` directly on the pipe descriptors. What is more interesting is to duplicate the pipe descriptors onto standard input or standard output. Often, the child then runs some other program, and that program can either read from its standard input (the pipe that we created) or write to its standard output (the pipe).

Example

Consider a program that displays some output that it has created, one page at a time. Rather than reinvent the pagination done by several UNIX system utilities, we want to invoke the user's favorite pager. To avoid writing all the data to a temporary file and calling `system` to display that file, we want to pipe the output directly to the pager. To do this, we create a pipe, `fork` a child process, set up the child's standard input to be the read end of the pipe, and `exec` the user's pager program. Figure 15.6 shows how to do this. (This example takes a command-line argument to specify the name of a file to display. Often, a program of this type would already have the data to display to the terminal in memory.)

```
#include "apue.h"
#include <sys/wait.h>

#define DEF_PAGER   "/bin/more"     /* default pager program */

int
main(int argc, char *argv[])
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    *pager, *argv0;
    char    line[MAXLINE];
    FILE    *fp;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");

    if ((fp = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);
    if (pipe(fd) < 0)
        err_sys("pipe error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {                                   /* parent */
        close(fd[0]);       /* close read end */

        /* parent copies argv[1] to pipe */
        while (fgets(line, MAXLINE, fp) != NULL) {
            n = strlen(line);
            if (write(fd[1], line, n) != n)
                err_sys("write error to pipe");
        }
        if (ferror(fp))
            err_sys("fgets error");

        close(fd[1]);   /* close write end of pipe for reader */

        if (waitpid(pid, NULL, 0) < 0)
            err_sys("waitpid error");
```

```
        exit(0);
    } else {                                        /* child */
        close(fd[1]);   /* close write end */
        if (fd[0] != STDIN_FILENO) {
            if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            close(fd[0]);   /* don't need this after dup2 */
        }

        /* get arguments for execl() */
        if ((pager = getenv("PAGER")) == NULL)
            pager = DEF_PAGER;
        if ((argv0 = strrchr(pager, '/')) != NULL)
            argv0++;        /* step past rightmost slash */
        else
            argv0 = pager;  /* no slash in pager */

        if (execl(pager, argv0, (char *)0) < 0)
            err_sys("execl error for %s", pager);
    }
    exit(0);
}
```

Figure 15.6 Copy file to pager program

Before calling `fork`, we create a pipe. After the `fork`, the parent closes its read end, and the child closes its write end. The child then calls `dup2` to have its standard input be the read end of the pipe. When the pager program is executed, its standard input will be the read end of the pipe.

When we duplicate one descriptor onto another (`fd[0]` onto standard input in the child), we have to be careful that the descriptor doesn't already have the desired value. If the descriptor already had the desired value and we called `dup2` and `close`, the single copy of the descriptor would be closed. (Recall the operation of `dup2` when its two arguments are equal, discussed in Section 3.12.) In this program, if standard input had not been opened by the shell, the `fopen` at the beginning of the program should have used descriptor 0, the lowest unused descriptor, so `fd[0]` should never equal standard input. Nevertheless, whenever we call `dup2` and `close` to duplicate one descriptor onto another, we'll always compare the descriptors first, as a defensive programming measure.

Note how we try to use the environment variable `PAGER` to obtain the name of the user's pager program. If this doesn't work, we use a default. This is a common usage of environment variables. □

Example

Recall the five functions `TELL_WAIT`, `TELL_PARENT`, `TELL_CHILD`, `WAIT_PARENT`, and `WAIT_CHILD` from Section 8.9. In Figure 10.24, we showed an implementation using signals. Figure 15.7 shows an implementation using pipes.

```
#include "apue.h"

static int  pfd1[2], pfd2[2];

void
TELL_WAIT(void)
{
    if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
        err_sys("pipe error");
}

void
TELL_PARENT(pid_t pid)
{
    if (write(pfd2[1], "c", 1) != 1)
        err_sys("write error");
}

void
WAIT_PARENT(void)
{
    char    c;

    if (read(pfd1[0], &c, 1) != 1)
        err_sys("read error");

    if (c != 'p')
        err_quit("WAIT_PARENT: incorrect data");
}

void
TELL_CHILD(pid_t pid)
{
    if (write(pfd1[1], "p", 1) != 1)
        err_sys("write error");
}

void
WAIT_CHILD(void)
{
    char    c;

    if (read(pfd2[0], &c, 1) != 1)
        err_sys("read error");

    if (c != 'c')
        err_quit("WAIT_CHILD: incorrect data");
}
```

Figure 15.7 Routines to let a parent and child synchronize