44

PIPES AND FIFOS

This chapter describes pipes and FIFOs. Pipes are the oldest method of IPC on the UNIX system, having appeared in Third Edition UNIX in the early 1970s. Pipes provide an elegant solution to a frequent requirement: having created two processes to run different programs (commands), how can the shell allow the output produced by one process to be used as the input to the other process? Pipes can be used to pass data between related processes (the meaning of *related* will become clear later). FIFOs are a variation on the pipe concept. The important difference is that FIFOs can be used for communication between *any* processes.

44.1 Overview

Every user of the shell is familiar with the use of pipes in commands such as the following, which counts the number of files in a directory:

```
$ ls | wc -l
```

In order to execute the above command, the shell creates two processes, executing *ls* and *wc*, respectively. (This is done using *fork()* and *exec()*, which are described in Chapters 24 and 27.) Figure 44-1 shows how the two processes employ the pipe.

Among other things, Figure 44-1 is intended to illustrate how pipes got their name. We can think of a pipe as a piece of plumbing that allows data to flow from one process to another.

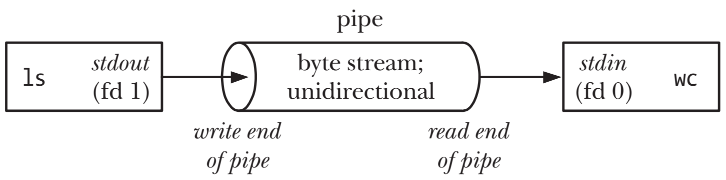

Figure 44-1: Using a pipe to connect two processes

One point to note in Figure 44-1 is that the two processes are connected to the pipe so that the writing process (*ls*) has its standard output (file descriptor 1) joined to the write end of the pipe, while the reading process (*wc*) has its standard input (file descriptor 0) joined to the read end of the pipe. In effect, these two processes are unaware of the existence of the pipe; they just read from and write to the standard file descriptors. The shell must do some work in order to set things up in this way, and we see how this is done in Section 44.4.

In the following paragraphs, we cover a number of important characteristics of pipes.

A pipe is a byte stream

When we say that a pipe is a byte stream, we mean that there is no concept of messages or message boundaries when using a pipe. The process reading from a pipe can read blocks of data of any size, regardless of the size of blocks written by the writing process. Furthermore, the data passes through the pipe sequentially—bytes are read from a pipe in exactly the order they were written. It is not possible to randomly access the data in a pipe using *lseek()*.

If we want to implement the notion of discrete messages in a pipe, we must do this within our application. While this is feasible (refer to Section 44.8), it may be preferable to use alternative IPC mechanisms, such as message queues and datagram sockets, which we discuss in later chapters.

Reading from a pipe

Attempts to read from a pipe that is currently empty block until at least one byte has been written to the pipe. If the write end of a pipe is closed, then a process reading from the pipe will see end-of-file (i.e., *read()* returns 0) once it has read all remaining data in the pipe.

Pipes are unidirectional

Data can travel only in one direction through a pipe. One end of the pipe is used for writing, and the other end is used for reading.

On some other UNIX implementations—notably those derived from System V Release 4—pipes are bidirectional (so-called *stream pipes*). Bidirectional pipes are not specified by any UNIX standards, so that, even on implementations where they are provided, it is best to avoid reliance on their semantics. As an alternative, we can use UNIX domain stream socket pairs (created using the *socketpair()* system call described in Section 57.5), which provide a standardized bidirectional communication mechanism that is semantically equivalent to stream pipes.

Writes of up to PIPE_BUF bytes are guaranteed to be atomic

If multiple processes are writing to a single pipe, then it is guaranteed that their data won't be intermingled if they write no more than PIPE_BUF bytes at a time.

SUSv3 requires that PIPE_BUF be at least _POSIX_PIPE_BUF (512). An implementation should define PIPE_BUF (in <limits.h>) and/or allow the call *fpathconf(fd, _PC_PIPE_BUF)* to return the actual upper limit for atomic writes. PIPE_BUF varies across UNIX implementations; for example, it is 512 bytes on FreeBSD 6.0, 4096 bytes on Tru64 5.1, and 5120 bytes on Solaris 8. On Linux, PIPE_BUF has the value 4096.

When writing blocks of data larger than PIPE_BUF bytes to a pipe, the kernel may transfer the data in multiple smaller pieces, appending further data as the reader removes bytes from the pipe. (The *write()* call blocks until all of the data has been written to the pipe.) When there is only a single process writing to a pipe (the usual case), this doesn't matter. However, if there are multiple writer processes, then writes of large blocks may be broken into segments of arbitrary size (which may be smaller than PIPE_BUF bytes) and interleaved with writes by other processes.

The PIPE_BUF limit affects exactly when data is transferred to the pipe. When writing up to PIPE_BUF bytes, *write()* will block if necessary until sufficient space is available in the pipe so that it can complete the operation atomically. When more than PIPE_BUF bytes are being written, *write()* transfers as much data as possible to fill the pipe, and then blocks until data has been removed from the pipe by some reading process. If such a blocked *write()* is interrupted by a signal handler, then the call unblocks and returns a count of the number of bytes successfully transferred, which will be less than was requested (a so-called *partial write*).

> On Linux 2.2, pipe writes of *any* size are atomic, unless interrupted by a signal handler. On Linux 2.4 and later, any write greater than PIPE_BUF bytes may be interleaved with writes by other processes. (The kernel code implementing pipes underwent substantial changes between kernel versions 2.2 and 2.4.)

Pipes have a limited capacity

A pipe is simply a buffer maintained in kernel memory. This buffer has a maximum capacity. Once a pipe is full, further writes to the pipe block until the reader removes some data from the pipe.

SUSv3 makes no requirement about the capacity of a pipe. In Linux kernels before 2.6.11, the pipe capacity is the same as the system page size (e.g., 4096 bytes on x86-32); since Linux 2.6.11, the pipe capacity is 65,536 bytes. Other UNIX implementations have different pipe capacities.

In general, an application never needs to know the exact capacity of a pipe. If we want to prevent the writer process(es) from blocking, the process(es) reading from the pipe should be designed to read data as soon as it is available.

> In theory, there is no reason why a pipe couldn't operate with smaller capacities, even with a single-byte buffer. The reason for employing large buffer sizes is efficiency: each time a writer fills the pipe, the kernel must perform a context switch to allow the reader to be scheduled so that it can empty some data from the pipe. Employing a larger buffer size means that fewer context switches are required.
>
> Starting with Linux 2.6.35, the capacity of a pipe can be modified. The Linux-specific call *fcntl(fd, F_SETPIPE_SZ, size)* changes the capacity of the pipe referred to by *fd* to be at least *size* bytes. An unprivileged process can

change the pipe capacity to any value in the range from the system page size up to the value in `/proc/sys/fs/pipe-max-size`. The default value for `pipe-max-size` is 1,048,576 bytes. A privileged (`CAP_SYS_RESOURCE`) process can override this limit. When allocating space for the pipe, the kernel may round *size* up to some value convenient for the implementation. The *fcntl(fd, F_GETPIPE_SZ)* call returns the actual size allocated for the pipe.

44.2 Creating and Using Pipes

The *pipe()* system call creates a new pipe.

```
#include <unistd.h>

int pipe(int filedes[2]);
                                        Returns 0 on success, or –1 on error
```

A successful call to *pipe()* returns two open file descriptors in the array *filedes*: one for the read end of the pipe (*filedes[0]*) and one for the write end (*filedes[1]*).

As with any file descriptor, we can use the *read()* and *write()* system calls to perform I/O on the pipe. Once written to the write end of a pipe, data is immediately available to be read from the read end. A *read()* from a pipe obtains the lesser of the number of bytes requested and the number of bytes currently available in the pipe (but blocks if the pipe is empty).

We can also use the *stdio* functions (*printf()*, *scanf()*, and so on) with pipes by first using *fdopen()* to obtain a file stream corresponding to one of the descriptors in *filedes* (Section 13.7). However, when doing this, we must be aware of the *stdio* buffering issues described in Section 44.6.

The call *ioctl(fd, FIONREAD, &cnt)* returns the number of unread bytes in the pipe or FIFO referred to by the file descriptor *fd*. This feature is also available on some other implementations, but is not specified in SUSv3.

Figure 44-2 shows the situation after a pipe has been created by *pipe()*, with the calling process having file descriptors referring to each end.

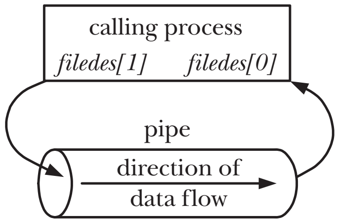

Figure 44-2: Process file descriptors after creating a pipe

A pipe has few uses within a single process (we consider one in Section 63.5.2). Normally, we use a pipe to allow communication between two processes. To connect two processes using a pipe, we follow the *pipe()* call with a call to *fork()*. During a *fork()*, the child process inherits copies of its parent's file descriptors (Section 24.2.1), bringing about the situation shown on the left side of Figure 44-3.

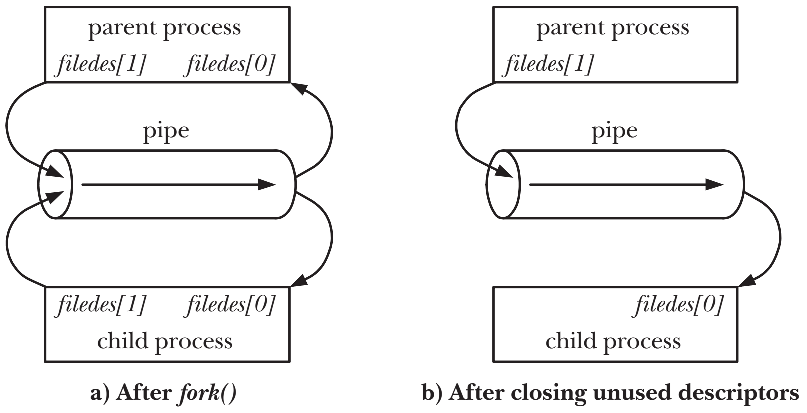

Figure 44-3: Setting up a pipe to transfer data from a parent to a child

While it is possible for the parent and child to both read from and write to the pipe, this is not usual. Therefore, immediately after the *fork()*, one process closes its descriptor for the write end of the pipe, and the other closes its descriptor for the read end. For example, if the parent is to send data to the child, then it would close its read descriptor for the pipe, *filedes[0]*, while the child would close its write descriptor for the pipe, *filedes[1]*, bringing about the situation shown on the right side of Figure 44-3. The code to create this setup is shown in Listing 44-1.

Listing 44-1: Steps in creating a pipe to transfer data from a parent to a child

```
    int filedes[2];

    if (pipe(filedes) == -1)                    /* Create the pipe */
        errExit("pipe");

    switch (fork()) {                           /* Create a child process */
    case -1:
        errExit("fork");

    case 0:  /* Child */
        if (close(filedes[1]) == -1)            /* Close unused write end */
            errExit("close");

        /* Child now reads from pipe */
        break;

    default: /* Parent */
        if (close(filedes[0]) == -1)            /* Close unused read end */
            errExit("close");

        /* Parent now writes to pipe */
        break;
    }
```

One reason that it is not usual to have both the parent and child reading from a single pipe is that if two processes try to simultaneously read from a pipe, we can't be sure which process will be the first to succeed—the two processes race for data. Preventing such races would require the use of some synchronization mechanism. However, if we require bidirectional communication, there is a simpler way: just create two pipes, one for sending data in each direction between the two processes. (If employing this technique, then we need to be wary of deadlocks that may occur if both processes block while trying to read from empty pipes or while trying to write to pipes that are already full.)

While it is possible to have multiple processes writing to a pipe, it is typical to have only a single writer. (We show one example of where it is useful to have multiple writers to a pipe in Section 44.3.) By contrast, there are situations where it can be useful to have multiple writers on a FIFO, and we see an example of this in Section 44.8.

> Starting with kernel 2.6.27, Linux supports a new, nonstandard system call, *pipe2()*. This system call performs the same task as *pipe()*, but supports an additional argument, *flags*, that can be used to modify the behavior of the system call. Two flags are supported. The `O_CLOEXEC` flag causes the kernel to enable the close-on-exec flag (`FD_CLOEXEC`) for the two new file descriptors. This flag is useful for the same reasons as the *open()* `O_CLOEXEC` flag described in Section 4.3.1. The `O_NONBLOCK` flag causes the kernel to mark both underlying open file descriptions as nonblocking, so that future I/O operations will be nonblocking. This saves additional calls to *fcntl()* to achieve the same result.

Pipes allow communication between related processes

In the discussion so far, we have talked about using pipes for communication between a parent and a child process. However, pipes can be used for communication between any two (or more) related processes, as long as the pipe was created by a common ancestor before the series of *fork()* calls that led to the existence of the processes. (This is what we meant when we referred to *related processes* at the beginning of this chapter.) For example, a pipe could be used for communication between a process and its grandchild. The first process creates the pipe, and then forks a child that in turn forks to yield the grandchild. A common scenario is that a pipe is used for communication between two siblings—their parent creates the pipe, and then creates the two children. This is what the shell does when building a pipeline.

> There is an exception to the statement that pipes can be used to communicate only between related processes. Passing a file descriptor over a UNIX domain socket (a technique that we briefly describe in Section 61.13.3) makes it possible to pass a file descriptor for a pipe to an unrelated process.

Closing unused pipe file descriptors

Closing unused pipe file descriptors is more than a matter of ensuring that a process doesn't exhaust its limited set of file descriptors—it is essential to the correct use of pipes. We now consider why the unused file descriptors for both the read and write ends of the pipe must be closed.

The process reading from the pipe closes its write descriptor for the pipe, so that, when the other process completes its output and closes its write descriptor, the reader sees end-of-file (once it has read any outstanding data in the pipe).

If the reading process doesn't close the write end of the pipe, then, after the other process closes its write descriptor, the reader won't see end-of-file, even after it has read all data from the pipe. Instead, a *read()* would block waiting for data, because the kernel knows that there is still at least one write descriptor open for the pipe. That this descriptor is held open by the reading process itself is irrelevant; in theory, that process could still write to the pipe, even if it is blocked trying to read. For example, the *read()* might be interrupted by a signal handler that writes data to the pipe. (This is a realistic scenario, as we'll see in Section 63.5.2.)

The writing process closes its read descriptor for the pipe for a different reason. When a process tries to write to a pipe for which no process has an open read descriptor, the kernel sends the `SIGPIPE` signal to the writing process. By default, this signal kills a process. A process can instead arrange to catch or ignore this signal, in which case the *write()* on the pipe fails with the error `EPIPE` (broken pipe). Receiving the `SIGPIPE` signal or getting the `EPIPE` error is a useful indication about the status of the pipe, and this is why unused read descriptors for the pipe should be closed.

> Note that the treatment of a *write()* that is interrupted by a `SIGPIPE` handler is special. Normally, when a *write()* (or other "slow" system call) is interrupted by a signal handler, the call is either automatically restarted or fails with the error `EINTR`, depending on whether the handler was installed with the *sigaction()* `SA_RESTART` flag (Section 21.5). The behavior in the case of `SIGPIPE` is different because it makes no sense either to automatically restart the *write()* or to simply indicate that the *write()* was interrupted by a handler (thus implying that the *write()* could usefully be manually restarted). In neither case can a subsequent *write()* attempt succeed, because the pipe will still be broken.

If the writing process doesn't close the read end of the pipe, then, even after the other process closes the read end of the pipe, the writing process will still be able to write to the pipe. Eventually, the writing process will fill the pipe, and a further attempt to write will block indefinitely.

One final reason for closing unused file descriptors is that it is only after all file descriptors in all processes that refer to a pipe are closed that the pipe is destroyed and its resources released for reuse by other processes. At this point, any unread data in the pipe is lost.

Example program

The program in Listing 44-2 demonstrates the use of a pipe for communication between parent and child processes. This example demonstrates the byte-stream nature of pipes referred to earlier—the parent writes its data in a single operation, while the child reads data from the pipe in small blocks.

The main program calls *pipe()* to create a pipe ①, and then calls *fork()* to create a child ②. After the *fork()*, the parent process closes its file descriptor for the read end of the pipe ⑧, and writes the string given as the program's command-line argument to the write end of the pipe ⑨. The parent then closes the read end of the pipe ⑩, and calls *wait()* to wait for the child to terminate ⑪. After closing its file descriptor for the write end of the pipe ③, the child process enters a loop where it reads ④ blocks of data (of up to `BUF_SIZE` bytes) from the pipe and writes ⑥ them to standard output. When the child encounters end-of-file on the pipe ⑤, it exits the loop ⑦, writes a trailing newline character, closes its descriptor for the read end of the pipe, and terminates.

Here's an example of what we might see when running the program in Listing 44-2:

```
$ ./simple_pipe 'It was a bright cold day in April, '\
'and the clocks were striking thirteen.'
It was a bright cold day in April, and the clocks were striking thirteen.
```

Listing 44-2: Using a pipe to communicate between a parent and child process

pipes/simple_pipe.c

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

   #define BUF_SIZE 10

   int
   main(int argc, char *argv[])
   {
       int pfd[2];                             /* Pipe file descriptors */
       char buf[BUF_SIZE];
       ssize_t numRead;

       if (argc != 2 || strcmp(argv[1], "--help") == 0)
           usageErr("%s string\n", argv[0]);

①      if (pipe(pfd) == -1)                    /* Create the pipe */
           errExit("pipe");

②      switch (fork()) {
       case -1:
           errExit("fork");

       case 0:             /* Child  - reads from pipe */
③          if (close(pfd[1]) == -1)            /* Write end is unused */
               errExit("close - child");

           for (;;) {              /* Read data from pipe, echo on stdout */
④              numRead = read(pfd[0], buf, BUF_SIZE);
               if (numRead == -1)
                   errExit("read");
⑤              if (numRead == 0)
                   break;                      /* End-of-file */
⑥              if (write(STDOUT_FILENO, buf, numRead) != numRead)
                   fatal("child - partial/failed write");
           }

⑦          write(STDOUT_FILENO, "\n", 1);
           if (close(pfd[0]) == -1)
               errExit("close");
           _exit(EXIT_SUCCESS);

       default:            /* Parent - writes to pipe */
⑧          if (close(pfd[0]) == -1)            /* Read end is unused */
               errExit("close - parent");
⑨          if (write(pfd[1], argv[1], strlen(argv[1])) != strlen(argv[1]))
               fatal("parent - partial/failed write");

⑩          if (close(pfd[1]) == -1)            /* Child will see EOF */
               errExit("close");
⑪          wait(NULL);                         /* Wait for child to finish */
           exit(EXIT_SUCCESS);
       }
   }
```

pipes/simple_pipe.c

44.3 Pipes as a Method of Process Synchronization

In Section 24.5, we looked at how signals could be used to synchronize the actions of parent and child processes in order to avoid race conditions. Pipes can be used to achieve a similar result, as shown by the skeleton program in Listing 44-3. This program creates multiple child processes (one for each command-line argument), each of which is intended to accomplish some action, simulated in the example program by sleeping for some time. The parent waits until all children have completed their actions.

To perform the synchronization, the parent builds a pipe ① before creating the child processes ②. Each child inherits a file descriptor for the write end of the pipe and closes this descriptor once it has completed its action ③. After all of the children have closed their file descriptors for the write end of the pipe, the parent's *read()* ⑤ from the pipe will complete, returning end-of-file (0). At this point, the parent is free to carry on to do other work. (Note that closing the unused write end of the pipe in the parent ④ is essential to the correct operation of this technique; otherwise, the parent would block forever when trying to read from the pipe.)

The following is an example of what we see when we use the program in Listing 44-3 to create three children that sleep for 4, 2, and 6 seconds:

```
$ ./pipe_sync 4 2 6
08:22:16  Parent started
08:22:18  Child 2 (PID=2445) closing pipe
08:22:20  Child 1 (PID=2444) closing pipe
08:22:22  Child 3 (PID=2446) closing pipe
08:22:22  Parent ready to go
```

Listing 44-3: Using a pipe to synchronize multiple processes

pipes/pipe_sync.c

```
#include "curr_time.h"                      /* Declaration of currTime() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int pfd[2];                             /* Process synchronization pipe */
    int j, dummy;
    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sleep-time...\n", argv[0]);

    setbuf(stdout, NULL);                   /* Make stdout unbuffered, since we
                                               terminate child with _exit() */
    printf("%s  Parent started\n", currTime("%T"));

①   if (pipe(pfd) == -1)
        errExit("pipe");

    for (j = 1; j < argc; j++) {
②       switch (fork()) {
        case -1:
            errExit("fork %d", j);

        case 0: /* Child */
            if (close(pfd[0]) == -1)        /* Read end is unused */
                errExit("close");

            /* Child does some work, and lets parent know it's done */

            sleep(getInt(argv[j], GN_NONNEG, "sleep-time"));
                                            /* Simulate processing */
            printf("%s  Child %d (PID=%ld) closing pipe\n",
                    currTime("%T"), j, (long) getpid());
③           if (close(pfd[1]) == -1)
                errExit("close");

            /* Child now carries on to do other things... */

            _exit(EXIT_SUCCESS);

        default: /* Parent loops to create next child */
            break;
        }
    }

    /* Parent comes here; close write end of pipe so we can see EOF */

④   if (close(pfd[1]) == -1)                /* Write end is unused */
        errExit("close");

    /* Parent may do other work, then synchronizes with children */

⑤   if (read(pfd[0], &dummy, 1) != 0)
        fatal("parent didn't get EOF");
    printf("%s  Parent ready to go\n", currTime("%T"));

    /* Parent can now carry on to do other things... */

    exit(EXIT_SUCCESS);
}
```
pipes/pipe_sync.c

Synchronization using pipes has an advantage over the earlier example of synchronization using signals: it can be used to coordinate the actions of one process with multiple other (related) processes. The fact that multiple (standard) signals can't be queued makes signals unsuitable in this case. (Conversely, signals have the advantage that they can be broadcast by one process to all of the members of a process group.)

Other synchronization topologies are possible (e.g., using multiple pipes). Furthermore, this technique could be extended so that, instead of closing the pipe, each child writes a message to the pipe containing its process ID and some status information. Alternatively, each child might write a single byte to the pipe. The parent process could then count and analyze these messages. This approach guards against the possibility of the child accidentally terminating, rather than explicitly closing the pipe.

44.4 Using Pipes to Connect Filters

When a pipe is created, the file descriptors used for the two ends of the pipe are the next lowest-numbered descriptors available. Since, in normal circumstances, descriptors 0, 1, and 2 are already in use for a process, some higher-numbered descriptors will be allocated for the pipe. So how do we bring about the situation shown in Figure 44-1, where two filters (i.e., programs that read from *stdin* and write to *stdout*) are connected using a pipe, such that the standard output of one program is directed into the pipe and the standard input of the other is taken from the pipe? And in particular, how can we do this without modifying the code of the filters themselves?

The answer is to use the techniques described in Section 5.5 for duplicating file descriptors. Traditionally, the following series of calls was used to accomplish the desired result:

```
int pfd[2];

pipe(pfd);          /* Allocates (say) file descriptors 3 and 4 for pipe */

/* Other steps here, e.g., fork() */

close(STDOUT_FILENO);           /* Free file descriptor 1 */
dup(pfd[1]);                    /* Duplication uses lowest free file
                                   descriptor, i.e., fd 1 */
```

The end result of the above steps is that the process's standard output is bound to the write end of the pipe. A corresponding set of calls can be used to bind a process's standard input to the read end of the pipe.

Note that these steps depend on the assumption that file descriptors 0, 1, and 2 for a process are already open. (The shell normally ensures this for each program it executes.) If file descriptor 0 was already closed prior to the above steps, then we would erroneously bind the process's standard *input* to the write end of the pipe. To avoid this possibility, we can replace the calls to *close()* and *dup()* with the following *dup2()* call, which allows us to explicitly specify the descriptor to be bound to the pipe end:

```
dup2(pfd[1], STDOUT_FILENO);    /* Close descriptor 1, and reopen bound
                                   to write end of pipe */
```

After duplicating *pfd[1]*, we now have two file descriptors referring to the write end of the pipe: descriptor 1 and *pfd[1]*. Since unused pipe file descriptors should be closed, after the *dup2()* call, we close the superfluous descriptor:

```
close(pfd[1]);
```

The code we have shown so far relies on standard output having been previously open. Suppose that, prior to the *pipe()* call, standard input and standard output had both been closed. In this case, *pipe()* would have allocated these two descriptors to the pipe, perhaps with *pfd[0]* having the value 0 and *pfd[1]* having the value 1. Consequently, the preceding *dup2()* and *close()* calls would be equivalent to the following:

```
dup2(1, 1);         /* Does nothing */
close(1);           /* Closes sole descriptor for write end of pipe */
```

Therefore, it is good defensive programming practice to bracket these calls with an `if` statement of the following form:

```
if (pfd[1] != STDOUT_FILENO) {
    dup2(pfd[1], STDOUT_FILENO);
    close(pfd[1]);
}
```

Example program

The program in Listing 44-4 uses the techniques described in this section to bring about the setup shown in Figure 44-1. After building a pipe, this program creates two child processes. The first child binds its standard output to the write end of the pipe and then execs *ls*. The second child binds its standard input to the read end of the pipe and then execs *wc*.

Listing 44-4: Using a pipe to connect *ls* and *wc*

pipes/pipe_ls_wc.c

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

int
main(int argc, char *argv[])
{
    int pfd[2];                                     /* Pipe file descriptors */

    if (pipe(pfd) == -1)                            /* Create pipe */
        errExit("pipe");

    switch (fork()) {
    case -1:
        errExit("fork");

    case 0:             /* First child: exec 'ls' to write to pipe */
        if (close(pfd[0]) == -1)                    /* Read end is unused */
            errExit("close 1");
        /* Duplicate stdout on write end of pipe; close duplicated descriptor */

        if (pfd[1] != STDOUT_FILENO) {              /* Defensive check */
            if (dup2(pfd[1], STDOUT_FILENO) == -1)
                errExit("dup2 1");
            if (close(pfd[1]) == -1)
                errExit("close 2");
        }

        execlp("ls", "ls", (char *) NULL);          /* Writes to pipe */
        errExit("execlp ls");

    default:            /* Parent falls through to create next child */
        break;
    }

    switch (fork()) {
    case -1:
        errExit("fork");

    case 0:             /* Second child: exec 'wc' to read from pipe */
        if (close(pfd[1]) == -1)                    /* Write end is unused */
            errExit("close 3");

        /* Duplicate stdin on read end of pipe; close duplicated descriptor */

        if (pfd[0] != STDIN_FILENO) {               /* Defensive check */
            if (dup2(pfd[0], STDIN_FILENO) == -1)
                errExit("dup2 2");
            if (close(pfd[0]) == -1)
                errExit("close 4");
        }

        execlp("wc", "wc", "-l", (char *) NULL);    /* Reads from pipe */
        errExit("execlp wc");

    default:            /* Parent falls through */
        break;
    }

    /* Parent closes unused file descriptors for pipe, and waits for children */

    if (close(pfd[0]) == -1)
        errExit("close 5");
    if (close(pfd[1]) == -1)
        errExit("close 6");
    if (wait(NULL) == -1)
        errExit("wait 1");
    if (wait(NULL) == -1)
        errExit("wait 2");

    exit(EXIT_SUCCESS);
}
```

pipes/pipe_ls_wc.c

When we run the program in Listing 44-4, we see the following:

```
$ ./pipe_ls_wc
     24
$ ls | wc -l                        Verify the results using shell commands
     24
```

44.5 Talking to a Shell Command via a Pipe: *popen()*

A common use for pipes is to execute a shell command and either read its output or send it some input. The *popen()* and *pclose()* functions are provided to simplify this task.

```
#include <stdio.h>

FILE *popen(const char *command, const char *mode);
                                        Returns file stream, or NULL on error
int pclose(FILE *stream);
                     Returns termination status of child process, or –1 on error
```

The *popen()* function creates a pipe, and then forks a child process that execs a shell, which in turn creates a child process to execute the string given in *command*. The *mode* argument is a string that determines whether the calling process will read from the pipe (*mode* is *r*) or write to it (*mode* is *w*). (Since pipes are unidirectional, two-way communication with the executed *command* is not possible.) The value of *mode* determines whether the standard output of the executed command is connected to the write end of the pipe or its standard input is connected to the read end of the pipe, as shown in Figure 44-4.

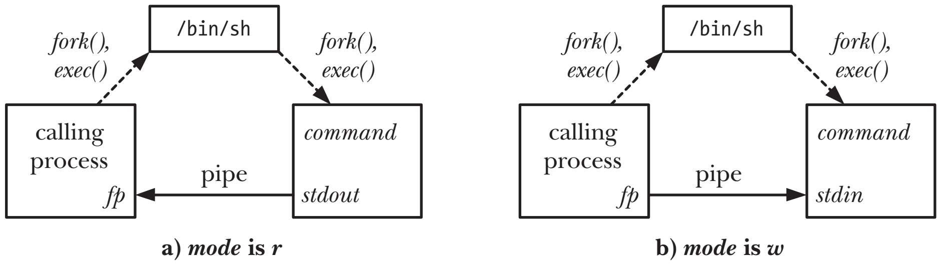

Figure 44-4: Overview of process relationships and pipe usage for *popen()*

On success, *popen()* returns a file stream pointer that can be used with the *stdio* library functions. If an error occurs (e.g., *mode* is not *r* or *w*, pipe creation fails, or the *fork()* to create the child fails), then *popen()* returns NULL and sets *errno* to indicate the cause of the error.

After the *popen()* call, the calling process uses the pipe to read the output of *command* or to send input to it. Just as with pipes created using *pipe()*, when reading from the pipe, the calling process encounters end-of-file once *command* has closed the write end of the pipe; when writing to the pipe, it is sent a SIGPIPE signal, and gets the EPIPE error, if *command* has closed the read end of the pipe.

Once I/O is complete, the *pclose()* function is used to close the pipe and wait for the child shell to terminate. (The *fclose()* function should not be used, since it doesn't wait for the child.) On success, *pclose()* yields the termination status (Section 26.1.3) of the child shell (which is the termination status of the last command that the shell executed, unless the shell was killed by a signal). As with *system()* (Section 27.6), if a shell could not be execed, then *pclose()* returns a value as though the child shell had terminated with the call *_exit(127)*. If some other error occurs, *pclose()* returns –1. One possible error is that the termination status could not be obtained. We explain how this may occur shortly.

When performing a wait to obtain the status of the child shell, SUSv3 requires that *pclose()*, like *system()*, should automatically restart the internal call that it makes to *waitpid()* if that call is interrupted by a signal handler.

In general, we can make the same statements for *popen()* as were made in Section 27.6 for *system()*. Using *popen()* offers convenience. It builds the pipe, performs descriptor duplication, closes unused descriptors, and handles all of the details of *fork()* and *exec()* on our behalf. In addition, shell processing is performed on the command. This convenience comes at the cost of efficiency. At least two extra processes must be created: one for the shell and one or more for the command(s) executed by the shell. As with *system()*, *popen()* should never be used from privileged programs.

While there are several similarities between *system()* and *popen()* plus *pclose()*, there are also significant differences. These stem from the fact that, with *system()*, the execution of the shell command is encapsulated within a single function call, whereas with *popen()*, the calling process runs in parallel with the shell command and then calls *pclose()*. The differences are as follows:

- Since the calling process and the executed command are operating in parallel, SUSv3 requires that *popen()* should *not* ignore SIGINT and SIGQUIT. If generated from the keyboard, these signals are sent to both the calling process and the executed command. This occurs because both processes reside in the same process group, and terminal-generated signals are sent to all of the members of the (foreground) process group, as described in Section 34.5.
- Since the calling process may create other child processes between the execution of *popen()* and *pclose()*, SUSv3 requires that *popen()* should *not* block SIGCHLD. This means that if the calling process performs a wait operation before the *pclose()* call, it may retrieve the status of the child created by *popen()*. In this case, when *pclose()* is later called, it will return –1, with *errno* set to ECHILD, indicating that *pclose()* could not retrieve the status of the child.

Example program

Listing 44-5 demonstrates the use of *popen()* and *pclose()*. This program repeatedly reads a filename wildcard pattern ②, and then uses *popen()* to obtain the results from passing this pattern to the *ls* command ⑤. (Techniques similar to this were used on older UNIX implementations to perform filename generation, also known as *globbing*, prior to the existence of the *glob()* library function.)

Listing 44-5: Globbing filename patterns with *popen()*

––– **pipes/popen_glob.c**

```
   #include <ctype.h>
   #include <limits.h>
   #include "print_wait_status.h"          /* For printWaitStatus() */
   #include "tlpi_hdr.h"

①  #define POPEN_FMT "/bin/ls -d %s 2> /dev/null"
   #define PAT_SIZE 50
   #define PCMD_BUF_SIZE (sizeof(POPEN_FMT) + PAT_SIZE)

   int
   main(int argc, char *argv[])
   {
       char pat[PAT_SIZE];                 /* Pattern for globbing */
       char popenCmd[PCMD_BUF_SIZE];
       FILE *fp;                           /* File stream returned by popen() */
       Boolean badPattern;                 /* Invalid characters in 'pat'? */
       int len, status, fileCnt, j;
       char pathname[PATH_MAX];

       for (;;) {                  /* Read pattern, display results of globbing */
           printf("pattern: ");
           fflush(stdout);
②          if (fgets(pat, PAT_SIZE, stdin) == NULL)
               break;                      /* EOF */
           len = strlen(pat);
           if (len <= 1)                   /* Empty line */
               continue;

           if (pat[len - 1] == '\n')       /* Strip trailing newline */
               pat[len - 1] = '\0';

           /* Ensure that the pattern contains only valid characters,
              i.e., letters, digits, underscore, dot, and the shell
              globbing characters. (Our definition of valid is more
              restrictive than the shell, which permits other characters
              to be included in a filename if they are quoted.) */

③          for (j = 0, badPattern = FALSE; j < len && !badPattern; j++)
               if (!isalnum((unsigned char) pat[j]) &&
                       strchr("_*?[^-].", pat[j]) == NULL)
                   badPattern = TRUE;

           if (badPattern) {
               printf("Bad pattern character: %c\n", pat[j - 1]);
               continue;
           }

           /* Build and execute command to glob 'pat' */

④          snprintf(popenCmd, PCMD_BUF_SIZE, POPEN_FMT, pat);
           popenCmd[PCMD_BUF_SIZE - 1] = '\0';     /* Ensure string is
                                                      null-terminated */
  ⑤        fp = popen(popenCmd, "r");
           if (fp == NULL) {
               printf("popen() failed\n");
               continue;
           }

           /* Read resulting list of pathnames until EOF */

           fileCnt = 0;
           while (fgets(pathname, PATH_MAX, fp) != NULL) {
               printf("%s", pathname);
               fileCnt++;
           }

           /* Close pipe, fetch and display termination status */

           status = pclose(fp);
           printf("    %d matching file%s\n", fileCnt, (fileCnt != 1) ? "s" : "");
           printf("    pclose() status == %#x\n", (unsigned int) status);
           if (status != -1)
               printWaitStatus("\t", status);
       }

       exit(EXIT_SUCCESS);
   }
```

pipes/popen_glob.c

The following shell session demonstrates the use of the program in Listing 44-5. In this example, we first provide a pattern that matches two filenames, and then a pattern that matches no filename:

```
$ ./popen_glob
pattern: popen_glob*                          Matches two filenames
popen_glob
popen_glob.c
    2 matching files
    pclose() status = 0
        child exited, status=0
pattern: x*                                   Matches no filename
    0 matching files
    pclose() status = 0x100                   ls(1) exits with status 1
        child exited, status=1
pattern: ^D$                                  Type Control-D to terminate
```

The construction of the command ①④ for globbing in Listing 44-5 requires some explanation. Actual globbing of a pattern is performed by the shell. The *ls* command is merely being used to list the matching filenames, one per line. We could have tried using the *echo* command instead, but this would have had the undesirable result that if a pattern matched no filenames, then the shell would leave the pattern unchanged, and *echo* would simply display the pattern. By contrast, if *ls* is given the name of a file that doesn't exist, it prints an error message on *stderr* (which we dispose of by redirecting *stderr* to `/dev/null`), prints nothing on *stdout*, and exits with a status of 1.

Note also the input checking performed in Listing 44-5 ③. This is done to prevent invalid input causing *popen()* to execute an unexpected shell command. Suppose that these checks were omitted, and the user entered the following input:

```
pattern: ; rm *
```

The program would then pass the following command to *popen()*, with disastrous results:

```
/bin/ls -d ; rm * 2> /dev/null
```

Such checking of input is always required in programs that use *popen()* (or *system()*) to execute a shell command built from user input. (An alternative would be for the application to quote any characters other than those being checked for, so that those characters don't undergo special processing by the shell.)

44.6 Pipes and *stdio* Buffering

Since the file stream pointer returned by a call to *popen()* doesn't refer to a terminal, the *stdio* library applies block buffering to the file stream (Section 13.2). This means that when we call *popen()* with a *mode* of *w*, then, by default, output is sent to the child process at the other end of the pipe only when the *stdio* buffer is filled or we close the pipe with *pclose()*. In many cases, this presents no problem. If, however, we need to ensure that the child process receives data on the pipe immediately, then we can either use periodic calls to *fflush()* or disable *stdio* buffering using the call *setbuf(fp, NULL)*. This technique can also be used if we create a pipe using the *pipe()* system call and then use *fdopen()* to obtain a *stdio* stream corresponding to the write end of the pipe.

If the process calling *popen()* is reading from the pipe (i.e., *mode* is *r*), things may not be so straightforward. In this case, if the child process is using the *stdio* library, then—unless it includes explicit calls to *fflush()* or *setbuf()*—its output will be available to the calling process only when the child either fills the *stdio* buffer or calls *fclose()*. (The same statement applies if we are reading from a pipe created using *pipe()* and the process writing on the other end is using the *stdio* library.) If this is a problem, there is little we can do unless we can modify the source code of the program running in the child process to include calls to *setbuf()* of *fflush()*.

If modifying the source code is not an option, then instead of using a pipe, we could use a pseudoterminal. A pseudoterminal is an IPC channel that appears to the process on one end as though it is a terminal. Consequently, the *stdio* library line buffers output. We describe pseudoterminals in Chapter 64.

44.7 FIFOs

Semantically, a FIFO is similar to a pipe. The principal difference is that a FIFO has a name within the file system and is opened in the same way as a regular file. This allows a FIFO to be used for communication between unrelated processes (e.g., a client and server).

Once a FIFO has been opened, we use the same I/O system calls as are used with pipes and other files (i.e., *read()*, *write()*, and *close()*). Just as with pipes, a FIFO has a write end and a read end, and data is read from the pipe in the same order as it is written. This fact gives FIFOs their name: *first in, first out*. FIFOs are also sometimes known as *named pipes*.

As with pipes, when all descriptors referring to a FIFO have been closed, any outstanding data is discarded.

We can create a FIFO from the shell using the *mkfifo* command:

```
$ mkfifo [ -m mode ] pathname
```

The *pathname* is the name of the FIFO to be created, and the *–m* option is used to specify a permission *mode* in the same way as for the *chmod* command.

When applied to a FIFO (or pipe), *fstat()* and *stat()* return a file type of `S_IFIFO` in the *st_mode* field of the *stat* structure (Section 15.1). When listed with *ls –l*, a FIFO is shown with the type *p* in the first column, and *ls –F* appends an the pipe symbol (|) to the FIFO pathname.

The *mkfifo()* function creates a new FIFO with the given *pathname*.

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

int mkfifo(const char *pathname, mode_t mode);
```

Returns 0 on success, or –1 on error

The *mode* argument specifies the permissions for the new FIFO. These permissions are specified by ORing the desired combination of constants from Table 15-4, on page 295. As usual, these permissions are masked against the process umask value (Section 15.4.6).

> Historically, FIFOs were created using the system call *mknod(pathname, S_IFIFO, 0)*. POSIX.1-1990 specified *mkfifo()* as a simpler API avoiding the generality of *mknod()*, which allows creation of various types of files, including device files. (SUSv3 specifies *mknod()*, but weakly, defining only its use for creating FIFOs.) Most UNIX implementations provide *mkfifo()* as a library function layered on top of *mknod()*.

Once a FIFO has been created, any process can open it, subject to the usual file permission checks (Section 15.4.3).

Opening a FIFO has somewhat unusual semantics. Generally, the only sensible use of a FIFO is to have a reading process and a writing process on each end. Therefore, by default, opening a FIFO for reading (the *open()* `O_RDONLY` flag) blocks until another process opens the FIFO for writing (the *open()* `O_WRONLY` flag). Conversely, opening the FIFO for writing blocks until another process opens the FIFO for reading. In other words, opening a FIFO synchronizes the reading and writing processes. If the opposite end of a FIFO is already open (perhaps because a pair of processes have already opened each end of the FIFO), then *open()* succeeds immediately.

Under most UNIX implementations (including Linux), it is possible to circumvent the blocking behavior when opening FIFOs by specifying the `O_RDWR` flag when opening a FIFO. In this case, *open()* returns immediately with a file descriptor that can be used for reading and writing on the FIFO. Doing this rather subverts the I/O model for FIFOs, and SUSv3 explicitly notes that opening a FIFO with the `O_RDWR` flag is unspecified; therefore, for portability reasons, this technique should be avoided. In circumstances where we need to prevent blocking when opening a FIFO, the *open()* `O_NONBLOCK` flag provides a standardized method for doing so (refer to Section 44.9).

> Avoiding the use of the `O_RDWR` flag when opening a FIFO can be desirable for a another reason. After such an *open()*, the calling process will never see end-of-file when reading from the resulting file descriptor, because there will always be at least one descriptor open for writing to the FIFO—the same descriptor from which the process is reading.

Using FIFOs and *tee(1)* to create a dual pipeline

One of the characteristics of shell pipelines is that they are linear; each process in the pipeline reads data produced by its predecessor and sends data to its successor. Using FIFOs, it is possible to create a fork in a pipeline, so that a duplicate copy of the output of a process is sent to another process in addition to its successor in the pipeline. In order to do this, we need to use the *tee* command, which writes two copies of what it reads from its standard input: one to standard output and the other to the file named in its command-line argument.

Making the *file* argument to *tee* a FIFO allows us to have two processes simultaneously reading the duplicate output produced by *tee*. We demonstrate this in the following shell session, which creates a FIFO named `myfifo`, starts a background *wc* command that opens the FIFO for reading (this will block until the FIFO is opened for writing), and then executes a pipeline that sends the output of *ls* to *tee*, which both passes the output further down the pipeline to *sort* and sends it to the `myfifo` FIFO. (The *–k5n* option to *sort* causes the output of *ls* to be sorted in increasing numerical order on the fifth space-delimited field.)

```
$ mkfifo myfifo
$ wc -l < myfifo &
$ ls -l | tee myfifo | sort -k5n
```
(Resulting output not shown)

Diagrammatically, the above commands create the situation shown in Figure 44-5.

> The *tee* program is so named because of its shape. We can consider *tee* as functioning similarly to a pipe, but with an additional branch that sends duplicate output. Diagrammatically, this has the shape of a capital letter *T* (see Figure 44-5). In addition to the purpose described here, *tee* is also useful for debugging pipelines and for saving the results produced at some intervening point in a complex pipeline.

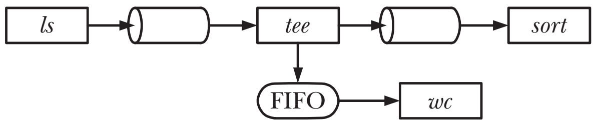

Figure 44-5: Using a FIFO and *tee(1)* to create a dual pipeline

44.8 A Client-Server Application Using FIFOs

In this section, we present a simple client-server application that employs FIFOs for IPC. The server provides the (trivial) service of assigning unique sequential numbers to each client that requests them. In the course of discussing this application, we introduce a few concepts and techniques in server design.

Application overview

In the example application, all clients send their requests to the server using a single server FIFO. The header file (Listing 44-6) defines the well-known name (`/tmp/seqnum_sv`) that the server uses for its FIFO. This name is fixed, so that all clients know how to contact the server. (In this example application, we create the FIFOs in the `/tmp` directory, since this allows us to conveniently run the programs without change on most systems. However, as noted in Section 38.7, creating files in publicly writable directories such as `/tmp` can lead to various security vulnerabilities and should be avoided in real-world applications.)

> In client-server applications, we'll repeatedly encounter the concept of a *well-known address* or name used by a server to make its service visible to clients. Using a well-known address is one solution to the problem of how clients can know where to contact a server. Another possible solution is to provide some kind of name server with which servers can register the names of their services. Each client then contacts the name server to obtain the location of the service it desires. This solution allows the location of servers to be flexible, at the cost of some extra programming effort. Of course, clients and servers then need to know where to contact the name server; typically, it resides at a well-known address.

It is not, however, possible to use a single FIFO to send responses to all clients, since multiple clients would race to read from the FIFO, and possibly read each other's response messages rather than their own. Therefore, each client creates a unique FIFO that the server uses for delivering the response for that client, and the server needs to know how to find each client's FIFO. One possible way to do this is for the client to generate its FIFO pathname, and then pass the pathname as part of its request message. Alternatively, the client and server can agree on a convention for constructing a client FIFO pathname, and, as part of its request, the client can pass the server the information required to construct the pathname specific to this client. This latter solution is used in our example. Each client's FIFO name is built from a template (`CLIENT_FIFO_TEMPLATE`) consisting of a pathname containing the client's process ID. The inclusion of the process ID provides an easy way of generating a name unique to this client.

Figure 44-6 shows how this application uses FIFOs for communication between the client and server processes of our application.

The header file (Listing 44-6) defines the formats for the request messages sent from clients to the server, and for the response messages sent from the server to clients.

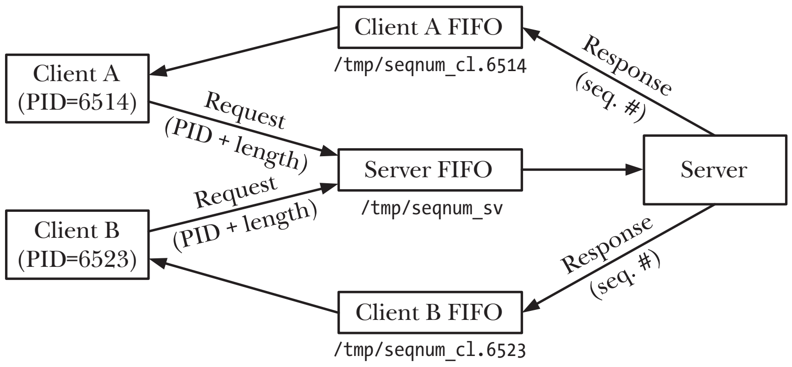

Figure 44-6: Using FIFOs in a single-server, multiple-client application

Recall that the data in pipes and FIFOs is a byte stream; boundaries between multiple messages are not preserved. This means that when multiple messages are being delivered to a single process, such as the server in our example, then the sender and receiver must agree on some convention for separating the messages. Various approaches are possible:

- Terminate each message with a *delimiter character*, such as a newline character. (For an example of this technique, see the *readLine()* function in Listing 59-1, on page 1201.) In this case, either the delimiter character must be one that never appears as part of the message, or we must adopt a convention for escaping the delimiter if it does occur within the message. For example, if we use a newline delimiter, then the characters \ plus newline could be used to represent a real newline character within the message, while \\ could represent a real \. One drawback of this approach is that the process reading messages must scan data from the FIFO a byte at a time until the delimiter character is found.
- Include a *fixed-size header with a length field* in each message specifying the number of bytes in the remaining variable-length component of the message. In this case, the reading process first reads the header from the FIFO, and then uses the header's length field to determine the number of bytes to read for the remainder of the message. This approach has the advantage of efficiently allowing messages of arbitrary size, but could lead to problems if a malformed message (e.g., bad *length* field) is written to the pipe.
- Use *fixed-length messages*, and have the server always read messages of this fixed size. This has the advantage of being simple to program. However, it places an upper limit on our message size and means that some channel capacity is wasted (since short messages must be padded to the fixed length). Furthermore, if one of the clients accidentally or deliberately sends a message that is not of the right length, then all subsequent messages will be out of step; in this situation, the server can't easily recover.

These three techniques are illustrated in Figure 44-7. Be aware that for each of these techniques, the total length of each message must be smaller than `PIPE_BUF` bytes in order to avoid the possibility of messages being broken up by the kernel and interleaved with messages from other writers.

In the three techniques described in the main text, a single channel (FIFO) is used for all messages from all clients. An alternative is to use a *single connection for each message*. The sender opens the communication channel, sends its message, and then closes the channel. The reading process knows that the message is complete when it encounters end-of-file. If multiple writers hold a FIFO open, then this approach is not feasible, because the reader won't see end-of-file when one of the writers closes the FIFO. This approach is, however, feasible when using stream sockets, where a server process creates a unique communication channel for each incoming client connection.

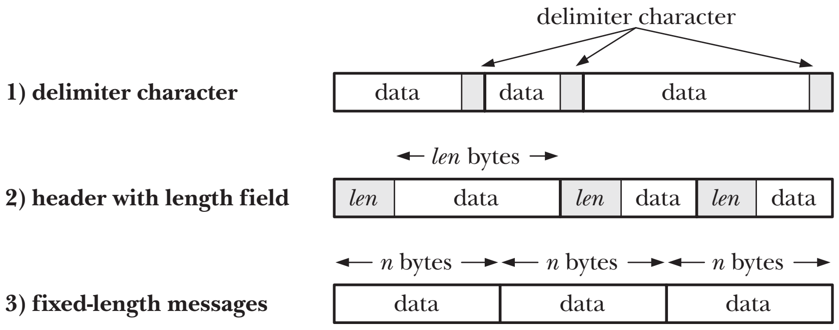

Figure 44-7: Separating messages in a byte stream

In our example application, we use the third of the techniques described above, with each client sending messages of a fixed size to the server. This message is defined by the *request* structure defined in Listing 44-6. Each request to the server includes the client's process ID, which enables the server to construct the name of the FIFO used by the client to receive a response. The request also contains a field (*seqLen*) specifying how many sequence numbers should be allocated to this client. The response message sent from server to client consists of a single field, *seqNum*, which is the starting value of the range of sequence numbers allocated to this client.

Listing 44-6: Header file for `fifo_seqnum_server.c` and `fifo_seqnum_client.c`

pipes/fifo_seqnum.h

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

#define SERVER_FIFO "/tmp/seqnum_sv"
                                /* Well-known name for server's FIFO */
#define CLIENT_FIFO_TEMPLATE "/tmp/seqnum_cl.%ld"
                                /* Template for building client FIFO name */
#define CLIENT_FIFO_NAME_LEN (sizeof(CLIENT_FIFO_TEMPLATE) + 20)
                                /* Space required for client FIFO pathname
                                   (+20 as a generous allowance for the PID) */

struct request {                /* Request (client --> server) */
    pid_t pid;                  /* PID of client */
    int seqLen;                 /* Length of desired sequence */
};
```

```
struct response {               /* Response (server --> client) */
    int seqNum;                 /* Start of sequence */
};
```
pipes/fifo_seqnum.h

Server program

Listing 44-7 is the code for the server. The server performs the following steps:

- Create the server's well-known FIFO ① and open the FIFO for reading ②. The server must be run before any clients, so that the server FIFO exists by the time a client attempts to open it. The server's *open()* blocks until the first client opens the other end of the server FIFO for writing.
- Open the server's FIFO once more ③, this time for writing. This will never block, since the FIFO has already been opened for reading. This second open is a convenience to ensure that the server doesn't see end-of-file if all clients close the write end of the FIFO.
- Ignore the `SIGPIPE` signal ④, so that if the server attempts to write to a client FIFO that doesn't have a reader, then, rather than being sent a `SIGPIPE` signal (which kills a process by default), it receives an `EPIPE` error from the *write()* system call.
- Enter a loop that reads and responds to each incoming client request ⑤. To send the response, the server constructs the name of the client FIFO ⑥ and then opens that FIFO ⑦.
- If the server encounters an error in opening the client FIFO, it abandons that client's request ⑧.

This is an example of an *iterative server*, in which the server reads and handles each client request before going on to handle the next client. An iterative server design is suitable when each client request can be quickly processed and responded to, so that other client requests are not delayed. An alternative design is a *concurrent server*, in which the main server process employs a separate child process (or thread) to handle each client request. We discuss server design further in Chapter 60.

Listing 44-7: An iterative server using FIFOs

pipes/fifo_seqnum_server.c
```
#include <signal.h>
#include "fifo_seqnum.h"

int
main(int argc, char *argv[])
{
    int serverFd, dummyFd, clientFd;
    char clientFifo[CLIENT_FIFO_NAME_LEN];
    struct request req;
    struct response resp;
    int seqNum = 0;                     /* This is our "service" */
```

```
       /* Create well-known FIFO, and open it for reading */

       umask(0);                           /* So we get the permissions we want */
①      if (mkfifo(SERVER_FIFO, S_IRUSR | S_IWUSR | S_IWGRP) == -1
               && errno != EEXIST)
           errExit("mkfifo %s", SERVER_FIFO);
②      serverFd = open(SERVER_FIFO, O_RDONLY);
       if (serverFd == -1)
           errExit("open %s", SERVER_FIFO);

       /* Open an extra write descriptor, so that we never see EOF */

③      dummyFd = open(SERVER_FIFO, O_WRONLY);
       if (dummyFd == -1)
           errExit("open %s", SERVER_FIFO);

④      if (signal(SIGPIPE, SIG_IGN) == SIG_ERR)
           errExit("signal");

⑤      for (;;) {                          /* Read requests and send responses */
           if (read(serverFd, &req, sizeof(struct request))
                   != sizeof(struct request)) {
               fprintf(stderr, "Error reading request; discarding\n");
               continue;                   /* Either partial read or error */
           }

           /* Open client FIFO (previously created by client) */

⑥          snprintf(clientFifo, CLIENT_FIFO_NAME_LEN, CLIENT_FIFO_TEMPLATE,
                   (long) req.pid);
⑦          clientFd = open(clientFifo, O_WRONLY);
           if (clientFd == -1) {           /* Open failed, give up on client */
               errMsg("open %s", clientFifo);
⑧              continue;
           }

           /* Send response and close FIFO */

           resp.seqNum = seqNum;
           if (write(clientFd, &resp, sizeof(struct response))
                   != sizeof(struct response))
               fprintf(stderr, "Error writing to FIFO %s\n", clientFifo);
           if (close(clientFd) == -1)
               errMsg("close");

           seqNum += req.seqLen;           /* Update our sequence number */
       }
   }
```
pipes/fifo_seqnum_server.c

Client program

Listing 44-8 is the code for the client. The client performs the following steps:

- Create a FIFO to be used for receiving a response from the server ②. This is done before sending the request, in order to ensure that the FIFO exists by the time the server attempts to open it and send a response message.
- Construct a message for the server containing the client's process ID and a number (taken from an optional command-line argument) specifying the length of the sequence that the client wishes the server to assign to it ④. (If no command-line argument is supplied, the default sequence length is 1.)
- Open the server FIFO ⑤ and send the message to the server ⑥.
- Open the client FIFO ⑦, and read and print the server's response ⑧.

The only other detail of note is the exit handler ①, established with *atexit()* ③, which ensures that the client's FIFO is deleted when the process exits. Alternatively, we could have simply placed an *unlink()* call immediately after the *open()* of the client FIFO. This would work because, at that point, after they have both performed blocking *open()* calls, the server and the client would each hold open file descriptors for the FIFO, and removing the FIFO name from the file system doesn't affect these descriptors or the open file descriptions to which they refer.

Here is an example of what we see when we run the client and server programs:

```
$ ./fifo_seqnum_server &
[1] 5066
$ ./fifo_seqnum_client 3            Request a sequence of three numbers
0                                   Assigned sequence begins at 0
$ ./fifo_seqnum_client 2            Request a sequence of two numbers
3                                   Assigned sequence begins at 3
$ ./fifo_seqnum_client              Request a single number
5
```

Listing 44-8: Client for the sequence-number server

pipes/fifo_seqnum_client.c

```
#include "fifo_seqnum.h"

static char clientFifo[CLIENT_FIFO_NAME_LEN];

static void             /* Invoked on exit to delete client FIFO */
① removeFifo(void)
{
    unlink(clientFifo);
}

int
main(int argc, char *argv[])
{
    int serverFd, clientFd;
    struct request req;
    struct response resp;
```

```
        if (argc > 1 && strcmp(argv[1], "--help") == 0)
            usageErr("%s [seq-len...]\n", argv[0]);

        /* Create our FIFO (before sending request, to avoid a race) */

        umask(0);                   /* So we get the permissions we want */
②       snprintf(clientFifo, CLIENT_FIFO_NAME_LEN, CLIENT_FIFO_TEMPLATE,
                (long) getpid());
        if (mkfifo(clientFifo, S_IRUSR | S_IWUSR | S_IWGRP) == -1
                    && errno != EEXIST)
            errExit("mkfifo %s", clientFifo);

③       if (atexit(removeFifo) != 0)
            errExit("atexit");

        /* Construct request message, open server FIFO, and send request */

④       req.pid = getpid();
        req.seqLen = (argc > 1) ? getInt(argv[1], GN_GT_0, "seq-len") : 1;

⑤       serverFd = open(SERVER_FIFO, O_WRONLY);
        if (serverFd == -1)
            errExit("open %s", SERVER_FIFO);

⑥       if (write(serverFd, &req, sizeof(struct request)) !=
                sizeof(struct request))
            fatal("Can't write to server");

        /* Open our FIFO, read and display response */

⑦       clientFd = open(clientFifo, O_RDONLY);
        if (clientFd == -1)
            errExit("open %s", clientFifo);

⑧       if (read(clientFd, &resp, sizeof(struct response))
                != sizeof(struct response))
            fatal("Can't read response from server");

        printf("%d\n", resp.seqNum);
        exit(EXIT_SUCCESS);
    }
```

pipes/fifo_seqnum_client.c

44.9 Nonblocking I/O

As noted earlier, when a process opens one end of a FIFO, it blocks if the other end of the FIFO has not yet been opened. Sometimes, it is desirable not to block, and for this purpose, the `O_NONBLOCK` flag can be specified when calling *open()*:

```
fd = open("fifopath", O_RDONLY | O_NONBLOCK);
if (fd == -1)
    errExit("open");
```

If the other end of the FIFO is already open, then the O_NONBLOCK flag has no effect on the *open()* call—it successfully opens the FIFO immediately, as usual. The O_NONBLOCK flag changes things only if the other end of the FIFO is not yet open, and the effect depends on whether we are opening the FIFO for reading or writing:

- If the FIFO is being opened for reading, and no process currently has the write end of the FIFO open, then the *open()* call succeeds immediately (just as though the other end of the FIFO was already open).
- If the FIFO is being opened FIFO for writing, and the other end of the FIFO is not already open for reading, then *open()* fails, setting *errno* to ENXIO.

The asymmetry of the O_NONBLOCK flag depending on whether the FIFO is being opened for reading or for writing can be explained as follows. It is okay to open a FIFO for reading when there is no writer at the other end of the FIFO, since any attempt to read from the FIFO simply returns no data. However, attempting to write to a FIFO for which there is no reader would result in the generation of the SIGPIPE signal and an EPIPE error from *write()*.

Table 44-1 summarizes the semantics of opening a FIFO, including the effects of O_NONBLOCK described above.

Table 44-1: Semantics of *open()* for a FIFO

Type of *open()*		**Result of *open()***	
open for	**additional flags**	**other end of FIFO open**	**other end of FIFO closed**
reading	none (blocking)	succeeds immediately	blocks
	O_NONBLOCK	succeeds immediately	succeeds immediately
writing	none (blocking)	succeeds immediately	blocks
	O_NONBLOCK	succeeds immediately	fails (ENXIO)

Using the O_NONBLOCK flag when opening a FIFO serves two main purposes:

- It allows a single process to open both ends of a FIFO. The process first opens the FIFO for reading specifying O_NONBLOCK, and then opens the FIFO for writing.
- It prevents deadlocks between processes opening two FIFOs.

A *deadlock* is a situation where two or more process are blocked because each is waiting on the other process(es) to complete some action. The two processes shown in Figure 44-8 are deadlocked. Each process is blocked waiting to open a FIFO for reading. This blocking would not happen if each process could perform its second step (opening the other FIFO for writing). This particular deadlock problem could be solved by reversing the order of steps 1 and 2 in process Y, while leaving the order in process X unchanged, or vice versa. However, such an arrangement of steps may not be easy to achieve in some applications. Instead, we can resolve the problem by having either process, or both, specify the O_NONBLOCK flag when opening the FIFOs for reading.

Process X	**Process Y**
1. Open FIFO A for reading *blocks*	1. Open FIFO B for reading *blocks*
2. Open FIFO B for writing	2. Open FIFO A for writing

Figure 44-8: Deadlock between processes opening two FIFOs

Nonblocking *read()* and *write()*

The `O_NONBLOCK` flag affects not only the semantics of *open()* but also—because the flag then remains set for the open file description—the semantics of subsequent *read()* and *write()* calls. We describe these effects in the next section.

Sometimes, we need to change the state of the `O_NONBLOCK` flag for a FIFO (or another type of file) that is already open. Scenarios where this need may arise include the following:

- We opened a FIFO using `O_NONBLOCK`, but we want subsequent *read()* and *write()* calls to operate in blocking mode.
- We want to enable nonblocking mode for a file descriptor that was returned by *pipe()*. More generally, we might want to change the nonblocking status of any file descriptor that was obtained other than from a call to *open()*—for example, one of the three standard descriptors that are automatically opened for each new program run by the shell or a file descriptor returned by *socket()*.
- For some application-specific purpose, we need to switch the setting of the `O_NONBLOCK` setting of a file descriptor on and off.

For these purposes, we can use *fcntl()* to enable or disable the `O_NONBLOCK` open file status flag. To enable the flag, we write the following (omitting error checking):

```
int flags;

flags = fcntl(fd, F_GETFL);       /* Fetch open files status flags */
flags |= O_NONBLOCK;              /* Enable O_NONBLOCK bit */
fcntl(fd, F_SETFL, flags);        /* Update open files status flags */
```

And to disable it, we write the following:

```
flags = fcntl(fd, F_GETFL);
flags &= ~O_NONBLOCK;             /* Disable O_NONBLOCK bit */
fcntl(fd, F_SETFL, flags);
```

44.10 Semantics of *read()* and *write()* on Pipes and FIFOs

Table 44-2 summarizes the operation of *read()* for pipes and FIFOs, and includes the effect of the `O_NONBLOCK` flag.

The only difference between blocking and nonblocking reads occurs when no data is present and the write end is open. In this case, a normal *read()* blocks, while a nonblocking *read()* fails with the error `EAGAIN`.

Table 44-2: Semantics of reading *n* bytes from a pipe or FIFO containing *p* bytes

O_NONBLOCK enabled?	Data bytes available in pipe or FIFO (*p*)			
	***p* = *0*, write end open**	***p* = *0*, write end closed**	***p* < *n***	***p* >= *n***
No	block	return 0 (EOF)	read *p* bytes	read *n* bytes
Yes	fail (EAGAIN)	return 0 (EOF)	read *p* bytes	read *n* bytes

The impact of the O_NONBLOCK flag when writing to a pipe or FIFO is made complex by interactions with the PIPE_BUF limit. The *write()* behavior is summarized in Table 44-3.

Table 44-3: Semantics of writing *n* bytes to a pipe or FIFO

O_NONBLOCK enabled?	Read end open		Read end closed
	n* <= *PIPE_BUF	***n* > *PIPE_BUF***	
No	Atomically write *n* bytes; may block until sufficient data is read for *write()* to be performed	Write *n* bytes; may block until sufficient data read for *write()* to complete; data may be interleaved with writes by other processes	SIGPIPE + EPIPE
Yes	If sufficient space is available to immediately write *n* bytes, then *write()* succeeds atomically; otherwise, it fails (EAGAIN)	If there is sufficient space to immediately write some bytes, then write between 1 and *n* bytes (which may be interleaved with data written by other processes); otherwise, *write()* fails (EAGAIN)	SIGPIPE + EPIPE

The O_NONBLOCK flag causes a *write()* on a pipe or FIFO to fail (with the error EAGAIN) in any case where data can't be transferred immediately. This means that if we are writing up to PIPE_BUF bytes, then the *write()* will fail if there is not sufficient space in the pipe or FIFO, because the kernel can't complete the operation immediately and can't perform a partial write, since that would break the requirement that writes of up to PIPE_BUF bytes are atomic.

When writing more than PIPE_BUF bytes at a time, a write is not required to be atomic. For this reason, *write()* transfers as many bytes as possible (a partial write) to fill up the pipe or FIFO. In this case, the return value from *write()* is the number of bytes actually transferred, and the caller must retry later in order to write the remaining bytes. However, if the pipe or FIFO is full, so that not even one byte can be transferred, then *write()* fails with the error EAGAIN.

44.11 Summary

Pipes were the first method of IPC under the UNIX system, and they are used frequently by the shell, as well as in other applications. A pipe is a unidirectional, limited-capacity byte stream that can be used for communication between related processes. Although blocks of data of any size can be written to a pipe, only writes that do not exceed PIPE_BUF bytes are guaranteed to be atomic. As well as being used as a method of IPC, pipes can also be used for process synchronization.

When using pipes, we must be careful to close unused descriptors in order to ensure that reading processes detect end-of-file and writing processes receive the `SIGPIPE` signal or the `EPIPE` error. (Usually, it is easiest to have the application writing to a pipe ignore `SIGPIPE` and detect a "broken" pipe via the `EPIPE` error.)

The *popen()* and *pclose()* functions allow a program to transfer data to or from a standard shell command, without needing to handle the details of creating a pipe, execing a shell, and closing unused file descriptors.

FIFOs operate in exactly the same way as pipes, except that they are created using *mkfifo()*, have a name in the file system, and can be opened by any process with appropriate permissions. By default, opening a FIFO for reading blocks until another process opens the FIFO for writing, and vice versa.

In the course of this chapter, we looked at a number of related topics. First, we saw how to duplicate file descriptors in such a manner that the standard input or output of a filter can be bound to a pipe. While presenting a client-server example using FIFOs, we touched on a number of topics in client-server design, including the use of a well-known address for a server and iterative versus concurrent server design. In developing the example FIFO application, we noted that, although data transmitted through a pipe is a byte stream, it is sometimes useful for communicating processes to package the data into messages, and we looked at various ways in which this could be accomplished.

Finally, we noted the effect of the `O_NONBLOCK` (nonblocking I/O) flag when opening and performing I/O on a FIFO. The `O_NONBLOCK` flag is useful if we don't want to block while opening a FIFO. It is also useful if we don't want reads to block if no data is available, or writes to block if there is insufficient space within a pipe or FIFO.

Further information

The implementation of pipes is discussed in [Bach, 1986] and [Bovet & Cesati, 2005]. Useful details about pipes and FIFOs can also be found in [Vahalia, 1996].

44.12 Exercises

44-1. Write a program that uses two pipes to enable bidirectional communication between a parent and child process. The parent process should loop reading a block of text from standard input and use one of the pipes to send the text to the child, which converts it to uppercase and sends it back to the parent via the other pipe. The parent reads the data coming back from the child and echoes it on standard output before continuing around the loop once more.

44-2. Implement *popen()* and *pclose()*. Although these functions are simplified by not requiring the signal handling employed in the implementation of *system()* (Section 27.7), you will need to be careful to correctly bind the pipe ends to file streams in each process, and to ensure that all unused descriptors referring to the pipe ends are closed. Since children created by multiple calls to *popen()* may be running at one time, you will need to maintain a data structure that associates the file stream pointers allocated by *popen()* with the corresponding child process IDs. (If using an array for this purpose, the value returned by the *fileno()* function, which obtains the file descriptor corresponding to a file stream, can be used to index the array.) Obtaining the correct process ID from this structure will allow *pclose()* to select the child upon which to wait. This structure will also assist with the SUSv3 requirement that any still-open file streams created by earlier calls to *popen()* must be closed in the new child process.

44-3. The server in Listing 44-7 (`fifo_seqnum_server.c`) always starts assigning sequence numbers from 0 each time it is started. Modify the program to use a backup file that is updated each time a sequence number is assigned. (The *open()* `O_SYNC` flag, described in Section 4.3.1, may be useful.) At startup, the program should check for the existence of this file, and if it is present, use the value it contains to initialize the sequence number. If the backup file can't be found on startup, the program should create a new file and start assigning sequence numbers beginning at 0. (An alternative to this technique would be to use memory-mapped files, described in Chapter 49.)

44-4. Add code to the server in Listing 44-7 (`fifo_seqnum_server.c`) so that if the program receives the `SIGINT` or `SIGTERM` signals, it removes the server FIFO and terminates.

44-5. The server in Listing 44-7 (`fifo_seqnum_server.c`) performs a second `O_WRONLY` open of the FIFO so that it never sees end-of-file when reading from the reading descriptor (*serverFd*) of the FIFO. Instead of doing this, an alternative approach could be tried: whenever the server sees end-of-file on the reading descriptor, it closes the descriptor, and then once more opens the FIFO for reading. (This open would block until the next client opened the FIFO for writing.) What is wrong with this approach?

44-6. The server in Listing 44-7 (`fifo_seqnum_server.c`) assumes that the client process is well behaved. If a misbehaving client created a client FIFO and sent a request to the server, but did not open its FIFO, then the server's attempt to open the client FIFO would block, and other client's requests would be indefinitely delayed. (If done maliciously, this would constitute a *denial-of-service attack*.) Devise a scheme to deal with this problem. Extend the server (and possibly the client in Listing 44-8) accordingly.

44-7. Write programs to verify the operation of nonblocking opens and nonblocking I/O on FIFOs (see Section 44.9).