12 Programming with Pipes

In This Chapter

- Review of the Pipe Model of IPC
- Differences Between Anonymous Pipes and Named Pipes
- Creating Anonymous and Named Pipes
- Communicating Through Pipes
- Command-Line Creation and Use of Pipes

Introduction

This chapter explores the GNU/Linux pipes. The pipe model is an older but still useful mechanism for interprocess communication. It looks at what are known as half-duplex pipes and also named pipes. Each offers a first-in-first-out (FIFO) queuing model to permit communication between processes.

The Pipe Model

One way to visualize a pipe is a one-way connector between two entities. For example, consider the following GNU/Linux command:

```
ls -1 | wc -l
```

This command creates two processes, one for the `ls -1` and another for `wc -l`. It then connects the two together by setting the standard-input of the second process to the standard-output of the first process (see Figure 12.1). This has the effect of counting the number of files in the current subdirectory.

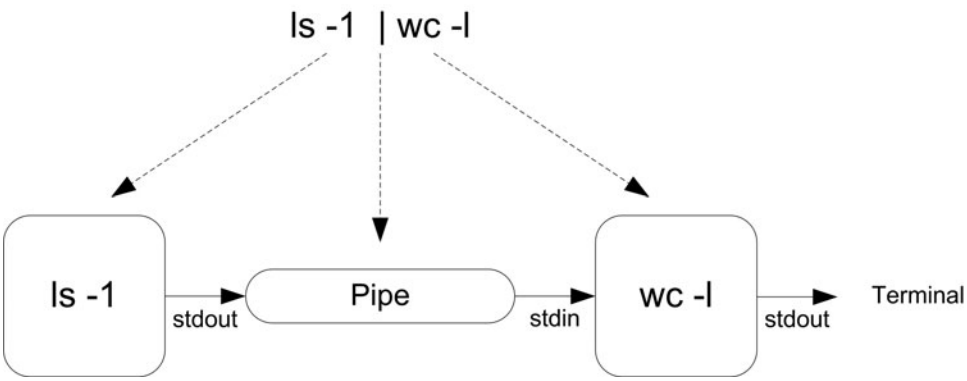

FIGURE 12.1 Simple pipe example.

This command, as illustrated in Figure 12.1, sets up a pipeline between two GNU/Linux commands. The `ls` command is performed, which generates output that is used as the input to the second command, `wc` (word count). This is a half-duplex pipe as communication occurs in one direction. The linkage between the two commands is facilitated by the GNU/Linux kernel, which takes care of connecting the two together. You can achieve this in applications as well, which this chapter demonstrates shortly.

Pipes and Named Pipes

A pipe, or half-duplex pipe, provides the means for a process to communicate with one of its ancestral subprocesses (of the anonymous variety). This is because no way exists in the operating system to locate the pipe (it's anonymous). Its most common use is to create a pipe at a parent process and then pass the pipe to the child so that they can communicate. Note that if full-duplex communication is required, the Sockets API should be considered instead.

Another type of pipe is called a *named pipe*. A named pipe works like a regular pipe but exists in the filesystem so that any process can find it. This means that processes not of the same ancestry are able to communicate with one another.

The following sections look at both half-duplex or anonymous pipes and named pipes. The chapter first takes a quick tour of pipes and then follows up with a more detailed look at the pipe API and GNU/Linux system-level commands that support pipes programming.

Whirlwind Tour

This section begins with a simple example of the pipe programming model. In this example, you create a pipe within a process, write a message to it, read the message back from the pipe, and then emit it (see Listing 12.1).

LISTING 12.1 Simple Pipe Example (on the CD-ROM at `./source/ch12/pipe1.c`)

```
1:      #include <unistd.h>
2:      #include <stdio.h>
3:      #include <string.h>
4:
5:      #define MAX_LINE        80
6:      #define PIPE_STDIN      0
7:      #define PIPE_STDOUT     1
8:
9:      int main()
10:     {
11:       const char *string={"A sample message."};
12:       int ret, myPipe[2];
13:       char buffer[MAX_LINE+1];
14:
15:       /* Create the pipe */
16:       ret = pipe( myPipe );
17:
18:       if (ret == 0) {
19:
20:         /* Write the message into the pipe */
21:         write( myPipe[PIPE_STDOUT], string, strlen(string) );
22:
23:         /* Read the message from the pipe */
24:         ret = read( myPipe[PIPE_STDIN], buffer, MAX_LINE );
25:
26:         /* Null terminate the string */
27:         buffer[ ret ] = 0;
28:
29:         printf("%s\n", buffer);
30:
31:       }
32:
33:       return 0;
34:     }
```

In Listing 12.1, you create your pipe using the `pipe` call at line 16. You pass in a two-element `int` array that represents your pipe. The pipe is defined as a pair of separate file descriptors, an input and an output. You can write to one end of the pipe and read from the other. The `pipe` API function returns zero on success. Upon return, the `myPipe` array contains two new file descriptors representing the input to the pipe (`myPipe[1]`) and the output from the pipe (`myPipe[0]`).

At line 21, you write your message to the pipe using the `write` function. You specify the `stdout` descriptor (from the perspective of the application, not the pipe). The pipe now contains the message and can be read at line 24 using the `read` function. Here again, from the perspective of the application, you use the `stdin` descriptor to read from the pipe. The `read` function stores what is read from the pipe in the `buffer` variable (argument three of the `read` function). You terminate it (add a `NULL` to the end) so that you can properly emit it at line 29 using `printf`. The pipe in this example is illustrated in Figure 12.2.

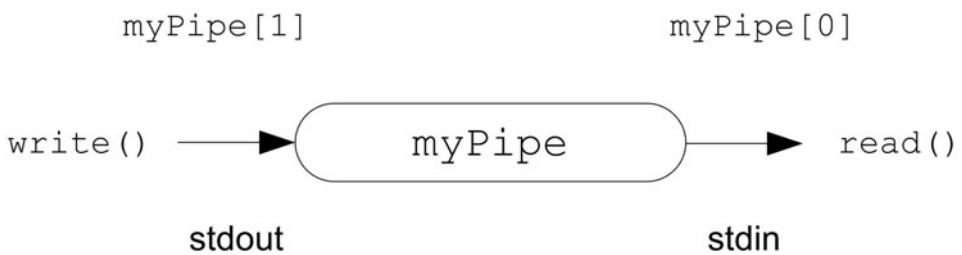

FIGURE 12.2 Half-duplex pipe example from Listing 12.1.

While this example was entertaining, communicating with yourself could be performed using any number of mechanisms. The detailed review looks at more complicated examples that provide communication between processes (both related and unrelated).

Detailed Review

While the `pipe` function is the majority of the pipe model, you need to understand a few other functions in their applicability toward pipe-based programming. Table 12.1 lists the functions that are detailed in this chapter.

This chapter also looks at some of the other functions that are applicable to pipe communication, specifically those that can be used to communicate using a pipe.

TABLE 12.1 API Functions for Pipe Programming

API Function	Use
`pipe`	Create a new anonymous pipe.
`dup`	Create a copy of a file descriptor.
`mkfifo`	Create a named pipe (`fifo`).

Remember that a pipe is nothing more than a pair of file descriptors, and therefore any functions that operate on file descriptors can be used. This includes but is not restricted to `select`, `read`, `write`, `fcntl`, `freopen`, *and such.*

pipe

The `pipe` API function creates a new pipe, represented by an array of two file descriptors. The `pipe` function has the following prototype:

```
#include <unistd.h>
int pipe( int fds[2] );
```

The `pipe` function returns 0 on success, or `-1` on failure, with `errno` set appropriately. On successful return, the `fds` array (which was passed by reference) is filled with two active file descriptors. The first element in the array is a file descriptor that can be read by the application, and the second element is a file descriptor that can be written to.

Now take a look at a slightly more complicated example of a pipe in a multiprocess application. In this application (see Listing 12.2), you create a pipe (line 14) and then fork your process into a parent and a child process (line 16). At the child, you attempt to read from the input file descriptor of your pipe (line 18), which suspends the process until something is available to read. When something is read, you terminate the string with a `NULL` and print out what was read. The parent simply writes a test string through the pipe using the `write` file descriptor (array offset 1 of the `pipe` structure) and then waits for the child to exit using the `wait` function.

Note that nothing is spectacular about this application except for the fact that the child process inherited the file descriptors that were created by the parent (using the `pipe` function) and then used them to communicate with one another. Recall that after the `fork` function is complete, the processes are independent (except that the child inherited features of the parent, such as the `pipe` file descriptors). Memory is separate, so the `pipe` method provides you with an interesting model to communicate between processes.

LISTING 12.2 Illustrating the Pipe Model with Two Processes (on the CD-ROM at `./source/ch12/fpipe.c`)

```
1:        #include <stdio.h>
2:        #include <unistd.h>
3:        #include <string.h>
4:        #include <wait.h>
5:
6:        #define MAX_LINE        80
7:
8:        int main()
9:        {
10:         int thePipe[2], ret;
11:         char buf[MAX_LINE+1];
12:         const char *testbuf={"a test string."};
13:
14:         if ( pipe( thePipe ) == 0 ) {
15:
16:           if (fork() == 0) {
17:
18:             ret = read( thePipe[0], buf, MAX_LINE );
19:             buf[ret] = 0;
20:             printf( "Child read %s\n", buf );
21:
22:           } else {
23:
24:             ret = write( thePipe[1], testbuf, strlen(testbuf) );
25:             ret = wait( NULL );
26:
27:           }
28:
29:         }
30:
31:         return 0;
32:       }
```

Note that so far these simple programs, have not discussed closing the pipe, because after the process finishes, the resources associated with the pipe are automatically freed. It's good programming practice, nonetheless, to close the descriptors of the pipe using the `close` call, as follows:

```
ret = pipe( myPipe );
...
close( myPipe[0] );
close( myPipe[1] );
```

If the write end of the pipe is closed and a process tries to read from the pipe, a zero is returned. This indicates that the pipe is no longer used and should be closed. If the read end of the pipe is closed and a process tries to write to it, a signal is generated. This signal (as discussed in Chapter 13, "Introduction to Sockets Programming") is called SIGPIPE. Applications that write to pipes commonly include a signal handler to catch just this situation.

dup AND dup2

The dup and dup2 calls are very useful functions that provide the ability to duplicate a file descriptor. They're most often used to redirect the stdin, stdout, or stderr of a process. The function prototypes for dup and dup2 are as follows:

```
#include <unistd.h>
int dup( int oldfd );
int dup2( int oldfd, int targetfd );
```

The dup function allows you to duplicate a descriptor. You pass in an existing descriptor, and it returns a new descriptor that is identical to the first. This means that both descriptors share the same internal structure. For example, if you perform an lseek (seek into the file) for one file descriptor, the file position is the same in the second. Use of the dup function is illustrated in the following code snippet:

```
int fd1, fd2;
...
fd2 = dup( fd1 );
```

Creating a descriptor prior to the fork call has the same effect as calling dup. The child process receives a duplicated descriptor, just like it would after calling dup.

The dup2 function is similar to dup but allows the caller to specify an active descriptor and the id of a target descriptor. Upon successful return of dup2, the new target descriptor is a duplicate of the first (targetfd = oldfd). Now take a look at a short code snippet that illustrates dup2:

```
int oldfd;
oldfd = open("app_log", (O_RDWR | O_CREATE), 0644 );
dup2( oldfd, 1 );
close( oldfd );
```

In this example, you open a new file called app_log and receive a file descriptor called fd1. You call dup2 with oldfd and 1, which has the effect of replacing the file

descriptor identified as `1` (`stdout`) with `oldfd` (the newly opened file). Anything written to `stdout` now goes instead to the file named `app_log`. Note that you close `oldfd` directly after duplicating it. This doesn't close your newly opened file, because file descriptor 1 now references it.

Now take a look at a more complex example. Recall that earlier in the chapter you investigated pipelining the output of `ls -1` to the input of `wc -l`. Now this example is explored in the context of a C application (see Listing 12.3).

You begin in Listing 12.3 by creating your pipe (line 9) and then forking the application into the child (lines 13–16) and parent (lines 20–23). In the child, we begin by closing the `stdout` descriptor (line 13). The child here provides the `ls -1` functionality and does not write to `stdout` but instead to the input to your pipe (redirected using `dup`). At line 14, you use `dup2` to redirect the `stdout` to your pipe (`pfds[1]`). After this is done, you close your input end of the pipe (as it will never be used). Finally, you use the `execlp` function to replace the child's image with that of the command `ls -1`. After this command executes, any output that is generated is sent to the input.

Now take a look at the receiving end of the pipe. The parent plays this role and follows a very similar pattern. You first close the `stdin` descriptor at line 20 (because you will accept nothing from it). Next, you use the `dup2` function again (line 21) to make the `stdin` the output end of the pipe. This is done by making file descriptor 0 (normal `stdin`) the same as `pfds[0]`. You close the `stdout` end of the pipe (`pfds[1]`) because you won't use it here (line 22). Finally, you `execlp` the command `wc -l`, which takes as its input the contents of the pipe (line 23).

Listing 12.3 Pipelining Commands in C (on the CD-ROM at `./source/ch12/dup.c`)

```
1:        #include <stdio.h>
2:        #include <stdlib.h>
3:        #include <unistd.h>
4:
5:        int main()
6:        {
7:          int pfds[2];
8:
9:          if ( pipe(pfds) == 0 ) {
10:
11:           if ( fork() == 0 ) {
12:
13:             close(1);
14:             dup2( pfds[1], 1 );
15:             close( pfds[0] );
16:             execlp( "ls", "ls", "-1", NULL );
```

```
17:
18:           } else {
19:
20:             close(0);
21:             dup2( pfds[0], 0 );
22:             close( pfds[1] );
23:             execlp( "wc", "wc", "-l", NULL );
24:
25:           }
26:
27:         }
28:
29:         return 0;
30:       }
```

What's important to note in this application is that your child process redirects its output to the input of the pipe, and the parent redirects its input to the output of the pipe—a very useful technique that is worth remembering.

mkfifo

The `mkfifo` function is used to create a file in the filesystem that provides FIFO functionality (otherwise known as a *named pipe*). Pipes that this chapter has discussed thus far are anonymous pipes. They're used exclusively between a process and its children. Named pipes are visible in the filesystem and therefore can be used by any (related or unrelated) process. The function prototype for `mkfifo` is defined as follows:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo( const char *pathname, mode_t mode );
```

The `mkfifo` command requires two arguments. The first (`pathname`) is the special file in the filesystem that is to be created. The second (`mode`) represents the read/write permissions for the FIFO. The `mkfifo` command returns 0 on success or `-1` on error (with `errno` filled appropriately). Take a look at an example of creating a `fifo` using the `mkfifo` function.

```
int ret;
...
ret = mkfifo( "/tmp/cmd_pipe", S_IFIFO | 0666 );
if (ret == 0) {
  // Named pipe successfully created
```

```
} else {
  // Failed to create named pipe
}
```

In this example, you create a `fifo` (named pipe) using the file `cmd_pipe` in the `/tmp` subdirectory. You can then open this file for `read` or `write` to communicate through it. After you open a named pipe, you can read from it using the typical I/O commands. For example, here's a snippet reading from the pipe using `fgets`:

```
pfp = fopen( "/tmp/cmd_pipe", "r" );
...
ret = fgets( buffer, MAX_LINE, pfp );
```

You can write to the pipe for this snippet using:

```
pfp = fopen( "/tmp/cmd_pipe", "w+ );
...
ret = fprintf( pfp, "Here's a test string!\n" );
```

What's interesting about named pipes, which is explored shortly in the discussion of the `mkfifo` system command, is that they work in what is known as a rendezvous model. A reader is unable to open the named pipe unless a writer has actively opened the other end of the pipe. The reader is blocked on the `open` call until a writer is present. Despite this limitation, the named pipe can be a useful mechanism for interprocess communication.

System Commands

Now it's time to take a look at a system command that is related to the pipe model for IPC. The `mkfifo` command, just like the `mkfifo` API function, allows you to create a named pipe from the command line.

mkfifo

The `mkfifo` command is one of two methods for creating a named pipe (`fifo` special file) at the command line. The general use of the `mkfifo` command is as follows:

```
mkfifo [options] name
```

where `[options]` are `-m` for mode (permissions) and `name` is the name of the named pipe to create (including path if needed). If permissions are not specified, the default is `0644`. Here's a sample use, creating a named pipe in `/tmp` called `cmd_pipe`:

```
$ mkfifo /tmp/cmd_pipe
```

You can adjust the options simply by specifying them with the `-m` option. Here's an example setting the permissions to `0644` (but deleting the original first):

```
$ rm cmd_pipe
$ mkfifo -m 0644 /tmp/cmd_pipe
```

After the permissions are created, you can communicate through this pipe via the command line. Consider the following scenario. In one terminal, you attempt to read from the pipe using the `cat` command:

```
$ cat cmd_pipe
```

Upon typing this command, you are suspended awaiting a writer opening the pipe. In another terminal, you write to the named pipe using the `echo` command, as follows:

```
$ echo Hi > cmd_pipe
```

When this command finishes, the reader wakes up and finishes (here's the complete reader command sequence again for clarity):

```
$ cat cmd_pipe
Hi
$
```

This illustrates that named pipes can be useful not only in C applications, but also in scripts (or combinations).

Named pipes can also be created with the `mknod` command (along with many other types of special files). You can create a named pipe (as with `mkfifo` before) as follows:

```
$ mknod cmd_pipe p
```

where the named pipe `cmd_pipe` is created in the current subdirectory (with type as `p` for named pipe).

Summary

This chapter was a very quick review of anonymous and named pipes. You reviewed application and command-line methods for creating pipes and also reviewed typical I/O mechanisms for communicating through them. You also reviewed the ability to redirect I/O using the `dup` and `dup2` commands. While useful for pipes, these commands are useful in many other scenarios as well (wherever a file descriptor is used, such as a socket or file).

Pipe Programming APIs

```
#include <unistd.h>
int pipe( int filedes[2] );
int dup( int oldfd );
int dup2( int oldfd, int targetfd );
int mkfifo( const char *pathname, mode_t mode );
```