5.4 Pipes

A *pipe* is a communication device that permits unidirectional communication. Data written to the "write end" of the pipe is read back from the "read end." Pipes are serial devices; the data is always read from the pipe in the same order it was written. Typically, a pipe is used to communicate between two threads in a single process or between parent and child processes.

In a shell, the symbol `|` creates a pipe. For example, this shell command causes the shell to produce two child processes, one for `ls` and one for `less`:

```
% ls | less
```

The shell also creates a pipe connecting the standard output of the `ls` subprocess with the standard input of the `less` process. The filenames listed by `ls` are sent to `less` in exactly the same order as if they were sent directly to the terminal.

A pipe's data capacity is limited. If the writer process writes faster than the reader process consumes the data, and if the pipe cannot store more data, the writer process blocks until more capacity becomes available. If the reader tries to read but no data is available, it blocks until data becomes available. Thus, the pipe automatically synchronizes the two processes.

5.4.1 Creating Pipes

To create a pipe, invoke the `pipe` command. Supply an integer array of size 2. The call to `pipe` stores the reading file descriptor in array position 0 and the writing file descriptor in position 1. For example, consider this code:

```
int pipe_fds[2];
int read_fd;
int write_fd;

pipe (pipe_fds);
read_fd = pipe_fds[0];
write_fd = pipe_fds[1];
```

Data written to the file descriptor `read_fd` can be read back from `write_fd`.

5.4.2 Communication Between Parent and Child Processes

A call to `pipe` creates file descriptors, which are valid only within that process and its children. A process's file descriptors cannot be passed to unrelated processes; however, when the process calls `fork`, file descriptors are copied to the new child process. Thus, pipes can connect only related processes.

In the program in Listing 5.7, a `fork` spawns a child process. The child inherits the pipe file descriptors. The parent writes a string to the pipe, and the child reads it out. The sample program converts these file descriptors into `FILE*` streams using `fdopen`. Because we use streams rather than file descriptors, we can use the higher-level standard C library I/O functions such as `printf` and `fgets`.

Listing 5.7 ***(pipe.c)* Using a Pipe to Communicate with a Child Process**

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

/* Write COUNT copies of MESSAGE to STREAM, pausing for a second
   between each.  */

void writer (const char* message, int count, FILE* stream)
{
  for (; count > 0; --count) {
    /* Write the message to the stream, and send it off immediately.  */
    fprintf (stream, "%s\n", message);
    fflush (stream);
    /* Snooze a while.  */
    sleep (1);
  }
}

/* Read random strings from the stream as long as possible.  */

void reader (FILE* stream)
{
  char buffer[1024];
  /* Read until we hit the end of the stream.  fgets reads until
     either a newline or the end-of-file.  */
  while (!feof (stream)
         && !ferror (stream)
         && fgets (buffer, sizeof (buffer), stream) != NULL)
    fputs (buffer, stdout);
}

int main ()
{
  int fds[2];
  pid_t pid;

  /* Create a pipe.  File descriptors for the two ends of the pipe are
     placed in fds.  */
  pipe (fds);
  /* Fork a child process.  */
  pid = fork ();
  if (pid == (pid_t) 0) {
    FILE* stream;
    /* This is the child process.  Close our copy of the write end of
       the file descriptor.  */
    close (fds[1]);
    /* Convert the read file descriptor to a FILE object, and read
       from it.  */
    stream = fdopen (fds[0], "r");
    reader (stream);
```

continues

Listing 5.7 **Continued**

```
      close (fds[0]);
    }
    else {
      /* This is the parent process.  */
      FILE* stream;
      /* Close our copy of the read end of the file descriptor.  */
      close (fds[0]);
      /* Convert the write file descriptor to a FILE object, and write
         to it.  */
      stream = fdopen (fds[1], "w");
      writer ("Hello, world.", 5, stream);
      close (fds[1]);
    }

    return 0;
  }
```

At the beginning of `main`, `fds` is declared to be an integer array with size 2. The `pipe` call creates a pipe and places the read and write file descriptors in that array. The program then forks a child process. After closing the read end of the pipe, the parent process starts writing strings to the pipe. After closing the write end of the pipe, the child reads strings from the pipe.

Note that after writing in the `writer` function, the parent flushes the pipe by calling `fflush`. Otherwise, the string may not be sent through the pipe immediately.

When you invoke the command `ls | less`, two forks occur: one for the `ls` child process and one for the `less` child process. Both of these processes inherit the pipe file descriptors so they can communicate using a pipe. To have unrelated processes communicate, use a FIFO instead, as discussed in Section 5.4.5, "FIFOs."

5.4.3 Redirecting the Standard Input, Output, and Error Streams

Frequently, you'll want to create a child process and set up one end of a pipe as its standard input or standard output. Using the `dup2` call, you can equate one file descriptor with another. For example, to redirect a process's standard input to a file descriptor `fd`, use this line:

```
dup2 (fd, STDIN_FILENO);
```

The symbolic constant `STDIN_FILENO` represents the file descriptor for the standard input, which has the value 0. The call closes standard input and then reopens it as a duplicate of `fd` so that the two may be used interchangeably. Equated file descriptors share the same file position and the same set of file status flags. Thus, characters read from `fd` are not reread from standard input.

The program in Listing 5.8 uses `dup2` to send the output from a pipe to the `sort` command.[2] After creating a pipe, the program forks. The parent process prints some strings to the pipe. The child process attaches the read file descriptor of the pipe to its standard input using `dup2`. It then executes the `sort` program.

Listing 5.8 **(*dup2.c*) Redirect Output from a Pipe with *dup2***

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main ()
{
  int fds[2];
  pid_t pid;

  /* Create a pipe.  File descriptors for the two ends of the pipe are
     placed in fds.  */
  pipe (fds);
  /* Fork a child process.  */
  pid = fork ();
  if (pid == (pid_t) 0) {
    /* This is the child process.  Close our copy of the write end of
       the file descriptor.  */
    close (fds[1]);
    /* Connect the read end of the pipe to standard input.  */
    dup2 (fds[0], STDIN_FILENO);
    /* Replace the child process with the "sort" program.  */
    execlp ("sort", "sort", 0);
  }
  else {
    /* This is the parent process.  */
    FILE* stream;
    /* Close our copy of the read end of the file descriptor.  */
    close (fds[0]);
    /* Convert the write file descriptor to a FILE object, and write
       to it.  */
    stream = fdopen (fds[1], "w");
    fprintf (stream, "This is a test.\n");
    fprintf (stream, "Hello, world.\n");
    fprintf (stream, "My dog has fleas.\n");
    fprintf (stream, "This program is great.\n");
    fprintf (stream, "One fish, two fish.\n");
    fflush (stream);
    close (fds[1]);
    /* Wait for the child process to finish.  */
    waitpid (pid, NULL, 0);
  }

  return 0;
}
```

2. `sort` reads lines of text from standard input, sorts them into alphabetical order, and prints them to standard output.

5.4.4 *popen* and *pclose*

A common use of pipes is to send data to or receive data from a program being run in a subprocess. The `popen` and `pclose` functions ease this paradigm by eliminating the need to invoke `pipe`, `fork`, `dup2`, `exec`, and `fdopen`.

Compare Listing 5.9, which uses `popen` and `pclose`, to the previous example (Listing 5.8).

Listing 5.9 **(*popen.c*) Example Using *popen***

```
#include <stdio.h>
#include <unistd.h>

int main ()
{
  FILE* stream = popen ("sort", "w");
  fprintf (stream, "This is a test.\n");
  fprintf (stream, "Hello, world.\n");
  fprintf (stream, "My dog has fleas.\n");
  fprintf (stream, "This program is great.\n");
  fprintf (stream, "One fish, two fish.\n");
  return pclose (stream);
}
```

The call to `popen` creates a child process executing the `sort` command, replacing calls to `pipe`, `fork`, `dup2`, and `execlp`. The second argument, `"w"`, indicates that this process wants to write to the child process. The return value from `popen` is one end of a pipe; the other end is connected to the child process's standard input. After the writing finishes, `pclose` closes the child process's stream, waits for the process to terminate, and returns its status value.

The first argument to `popen` is executed as a shell command in a subprocess running `/bin/sh`. The shell searches the `PATH` environment variable in the usual way to find programs to execute. If the second argument is `"r"`, the function returns the child process's standard output stream so that the parent can read the output. If the second argument is `"w"`, the function returns the child process's standard input stream so that the parent can send data. If an error occurs, `popen` returns a null pointer.

Call `pclose` to close a stream returned by `popen`. After closing the specified stream, `pclose` waits for the child process to terminate.

5.4.5 FIFOs

A *first-in, first-out (FIFO)* file is a pipe that has a name in the filesystem. Any process can open or close the FIFO; the processes on either end of the pipe need not be related to each other. FIFOs are also called *named pipes*.

You can make a FIFO using the `mkfifo` command. Specify the path to the FIFO on the command line. For example, create a FIFO in `/tmp/fifo` by invoking this:

```
% mkfifo /tmp/fifo
% ls -l /tmp/fifo
prw-rw-rw-    1 samuel   users          0 Jan 16 14:04 /tmp/fifo
```

The first character of the output from `ls` is `p`, indicating that this file is actually a FIFO (named pipe). In one window, read from the FIFO by invoking the following:

```
% cat < /tmp/fifo
```

In a second window, write to the FIFO by invoking this:

```
% cat > /tmp/fifo
```

Then type in some lines of text. Each time you press Enter, the line of text is sent through the FIFO and appears in the first window. Close the FIFO by pressing Ctrl+D in the second window. Remove the FIFO with this line:

```
% rm /tmp/fifo
```

Creating a FIFO

Create a FIFO programmatically using the `mkfifo` function. The first argument is the path at which to create the FIFO; the second parameter specifies the pipe's owner, group, and world permissions, as discussed in Chapter 10, "Security," Section 10.3, "File System Permissions." Because a pipe must have a reader and a writer, the permissions must include both read and write permissions. If the pipe cannot be created (for instance, if a file with that name already exists), `mkfifo` returns –1. Include `<sys/types.h>` and `<sys/stat.h>` if you call `mkfifo`.

Accessing a FIFO

Access a FIFO just like an ordinary file. To communicate through a FIFO, one program must open it for writing, and another program must open it for reading. Either low-level I/O functions (`open`, `write`, `read`, `close`, and so on, as listed in Appendix B, "Low-Level I/O") or C library I/O functions (`fopen`, `fprintf`, `fscanf`, `fclose`, and so on) may be used.

For example, to write a buffer of data to a FIFO using low-level I/O routines, you could use this code:

```
int fd = open (fifo_path, O_WRONLY);
write (fd, data, data_length);
close (fd);
```

To read a string from the FIFO using C library I/O functions, you could use this code:

```
FILE* fifo = fopen (fifo_path, "r");
fscanf (fifo, "%s", buffer);
fclose (fifo);
```

A FIFO can have multiple readers or multiple writers. Bytes from each writer are written atomically up to a maximum size of `PIPE_BUF` (4KB on Linux). Chunks from simultaneous writers can be interleaved. Similar rules apply to simultaneous reads.

Differences from Windows Named Pipes

Pipes in the Win32 operating systems are very similar to Linux pipes. (Refer to the Win32 library documentation for technical details about these.) The main differences concern named pipes, which, for Win32, function more like sockets. Win32 named pipes can connect processes on separate computers connected via a network. On Linux, sockets are used for this purpose. Also, Win32 allows multiple reader-writer connections on a named pipe without interleaving data, and pipes can be used for two-way communication.[3]

5.5 Sockets

A *socket* is a bidirectional communication device that can be used to communicate with another process on the same machine or with a process running on other machines. Sockets are the only interprocess communication we'll discuss in this chapter that permit communication between processes on different computers. Internet programs such as Telnet, rlogin, FTP, talk, and the World Wide Web use sockets.

For example, you can obtain the WWW page from a Web server using the Telnet program because they both use sockets for network communications.[4] To open a connection to a WWW server at `www.codesourcery.com`, use `telnet www.codesourcery.com 80`. The magic constant 80 specifies a connection to the Web server programming running `www.codesourcery.com` instead of some other process. Try typing `GET /` after the connection is established. This sends a message through the socket to the Web server, which replies by sending the home page's HTML source and then closing the connection—for example:

```
% telnet www.codesourcery.com 80
Trying 206.168.99.1...
Connected to merlin.codesourcery.com (206.168.99.1).
Escape character is '^]'.
GET /
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
...
```

3. Note that only Windows NT can create a named pipe; Windows 9*x* programs can form only client connections.

4. Usually, you'd use `telnet` to connect a Telnet server for remote logins. But you can also use `telnet` to connect to a server of a different kind and then type comments directly at it.