Chapter 5

PIPES

5.1 Introduction

We have discussed the nature and generation of processes. In the previous chapter we addressed primitive techniques for communicating between two or more processes. These techniques were limited in scope and suffered from a lack of reliable synchronization. Beginning with this chapter, we explore interprocess communication techniques using system-designed interprocess facilities. We start with **pipes**, which provide processes with a simple, synchronized way of passing information. By the early 1970s pipes became a standard part of UNIX.

We can think of the pipe as a special file that can store a limited amount of data in a first in, first out (FIFO) manner. On most systems, pipes are limited to a specific size. In Linux, the defined constant PIPE_SIZE (which is usually equivalent to the PAGE_SIZE for the system) establishes the total number of bytes allocated for a pipe. The defined constant PIPE_BUF (found in `<linux/limits.h>`, which is included by `<limits.h>`) sets the block size for an atomic write to a pipe. On our system the value for PIPE_BUF is 4096. Generally, one process writes to the pipe (as if it were a file), while another process reads from the pipe.

As shown in Figure 5.1, conceptually we can envision the pipe as a conveyor belt composed of data blocks that are continuously filled at (written to) the "write end" and emptied (read) from the "read end." The system keeps track of the current location of the last read/write location. Data is written to one end of the pipe and read from the other. From an implementation standpoint, an actual file pointer (as associated with a regular file) is not defined for a pipe, and as such no seeking is supported.

Figure 5.1 Conceptual data access using a pipe.

The operating system provides the synchronization between the writing and reading processes. By default, if a writing process attempts to **`write`** to a full pipe, the system automatically blocks the process until the pipe is able to receive the data. Likewise, if a **`read`** is attempted on an empty pipe, the process blocks until data is available. In addition, the process blocks if a specified pipe has been opened for reading, but another process has not opened the pipe for writing.

In a program, data is written to the pipe using the unbuffered I/O **`write`** system call (Table 5.1).

Table 5.1 Summary of the `write` System Call.

Include File(s)	`<unistd.h>`		Manual Section	**2**
Summary	`ssize_t write(int fd, const void *buf, size_t count);`			
Return	Success	Failure	Sets `errno`	
	Number of bytes written	−1	Yes	

Using the file descriptor specified by `fd`, the **`write`** system call attempts to write `count` bytes from the buffer referenced by `buf`. If the **`write`** system call is successful, the number of bytes actually written is returned. Otherwise,

Table 5.2 `write` Error Messages.

#	Constant	`perror` Message	Explanation
4	EINTR	Interrupted system call	Signal was caught during the system call.
5	EIO	I/O error	Low-level I/O error while attempting **`read`** from or **`write`** to file system.
6	ENXIO	No such device or address	O_NONBLOCK | O_WRONLY is set, the named file is a FIFO, and no process has the file open for reading.
9	EBADF	Bad file descriptor	`fd` is an invalid file descriptor or is not open for writing.
11	EAGAIN	Resource temporarily unavailable	• O_NDELAY or O_NONBLOCK is set and the file is currently locked by another process. • System memory for raw I/O is temporarily insufficient. • Attempted a **`write`** to pipe of `count` bytes, but less than `count` bytes is available.
14	EFAULT	Bad address	`buf` references an illegal address.
22	EINVAL	Invalid argument	`fd` associated with an object unsuitable for writing.
27	EFBIG	File too large	Attempt to **`write`** to a file that exceeds the current system limits.
28	ENOSPC	No space left on device	Device with file has run out of room.
32	EPIPE	Broken pipe	• Attempt to `write` to a pipe that is not opened for reading on one end (in this case a SIGPIPE signal also generated). • Attempt to `write` to a FIFO that is not opened for reading on one end. • Attempt to **`write`** to a pipe with only one end open.
34	ERANGE	Numerical result out of range	`count` value is less than 0 or greater than system limit.
35	EDEADLK	Resource deadlock avoided	The **`write`** system call would have gone to sleep generating a deadlock situation.
37	ENOLCK	No locks available	• Locking enabled, but region was previously locked. • System lock table is full.
63	ENOSR	Out of streams resources	Attempt to **`write`** to a stream, but insufficient stream memory is available.
67	ENOLINK	The link has been severed	The `buf` value references a remote system that is no longer active.

a –1 is returned and the global variable `errno` is set to indicate the nature of the error. As shown in Table 5.2, the number of ways in which **`write`** can fail is impressive indeed!

`write`s to a pipe are similar to those for a file except that

- Each file **`write`** request is always *appended* to the end of the pipe.
- **`write`** requests of PIPE_BUF size or less are guaranteed to not be interleaved with other **`write`** requests to the same pipe.[1]
- When the O_NONBLOCK and O_NDELAY flags are clear, a **`write`** request may cause the process to block. The defined constants O_NONBLOCK and O_NDELAY are included by the header file `<sys/fcntl.h>` and can be set with the **`fcntl`** system call. By default, these values are considered to be cleared, thus **`write`** blocks if the device is busy and **`write`**s are delayed (written to an internal buffer, which is written out to disk by the kernel at a later time). Once the **`write`** has completed, it returns the number of bytes successfully written.
- When the O_NONBLOCK or O_NDELAY flags are set and the request to **`write`** PIPE_BUF bytes or less is not successful, the value returned by the **`write`** system call can be summarized as

O_NONBLOCK	O_NDELAY	Value Returned
set	clear	–1
clear	set	0

  If both O_NONBLOCK and O_NDELAY flags are set, **`write`** will not block the process.
- If a **`write`** is made to a pipe that is not open for reading by any process, a SIGPIPE signal is generated and the value in `errno` is set to EPIPE (broken pipe). The default action (if not caught) for the SIGPIPE signal is termination.

Data is read from the pipe using the unbuffered I/O **`read`** system call summarized in Table 5.3.

[1] While **`write`** may still work if the number of bytes is greater than PIPE_BUF, it is best to stay within this limitation to guarantee the integrity of data.

Table 5.3 Summary of the `read` System Call.

Include File(s)	`<unistd.h>`		Manual Section	**2**
Summary	`ssize_t read(int fd, void *buf, size_t count);`			
Return	Success	Failure	Sets `errno`	
	Number of bytes read	−1	Yes	

The **`read`** system call reads `count` bytes from the open file associated with the file descriptor `fd` into the buffer referenced by `buf`. If the **`read`** call is successful, the number of bytes actually read is returned. If the number of bytes left in the pipe is less than `count`, the value returned by **`read`** will reflect this. When at the end of the file, a value of 0 is returned. If the **`read`** system call fails, a –1 is returned and the global variable `errno` is set. The values that `errno` may take when **`read`** fails are shown in Table 5.4.

Table 5.4 `read` Error Messages.

#	Constant	`perror` Message	Explanation
4	EINTR	Interrupted system call	Signal was caught during the system call.
5	EIO	I/O error	Background process cannot **`read`** from its controlling terminal.
6	ENXIO	No such device or address	File descriptor reference is invalid.
9	EBADF	Bad file descriptor	`fd` is an invalid file or is not open for reading.
11	EAGAIN	Resource temporarily unavailable	• O_NDELAY or O_NONBLOCK is set, and the file is currently locked by another process. • System memory for raw I/O is temporarily insufficient. • O_NDELAY or O_NONBLOCK is set, but there is no data waiting to be **`read`**.
14	EFAULT	Bad address	`buf` references an illegal address.
22	EINVAL	Invalid argument	`fd` associated with an unsuitable object for reading.

Continued

Table 5.4 *(Continued)*

#	Constant	`perror` Message	Explanation
35	EDEADLK	Resource deadlock avoided	The **`read`** system call would have gone to sleep generating a deadlock situation.
37	ENOLCK	No locks available	• Locking enabled, but region was previously locked. • System lock table is full.
67	ENOLINK	Link has been severed	The `buf` value references a remote system that is no longer active.
74	EBADMSG	Not a data message	Message to be **`read`** is not a data message.

In other aspects, **`read`**s performed on a pipe are similar to those on a file except that

- All **`reads`** are initiated from the current position (i.e., no seeking is supported).
- If both O_NONBLOCK and O_NDELAY flags are clear, then a **`read`** system call blocks (by default) until data is written to the pipe or the pipe is closed.
- If the pipe is **`open`** for writing by another process, but the pipe is empty, then a **`read`** (in combination with the flags O_NDELAY and O_NONBLOCK) will return the values

O_NONBLOCK	O_NDELAY	Value Returned
set	clear	–1
clear	set	0

- If the pipe is not opened for writing by another process, **`read`** returns a 0 (indicating the end-of-file condition). Note, this is the same value that is returned when the O_NDELAY flag has been set, and the pipe is open but empty.

Pipes can be divided into two categories: **unnamed** pipes and **named** pipes. Unnamed pipes can be used only with related processes (e.g., parent/child or child/child) and exist only for as long as the processes using them exist. Named pipes actually exist as directory entries. As such, they have file access permissions and can be used with unrelated processes.

5.2 Unnamed Pipes

An unnamed pipe is constructed with the **pipe** system call (see Table 5.5).

Table 5.5 Summary of the `pipe` System Call.

Include File(s)	`<unistd.h>`		Manual Section	**2**
Summary	`int pipe(int filedes[2]);`			
Return	Success	Failure	Sets `errno`	
	0	–1	Yes	

If successful, the **pipe** system call returns a pair of integer file descriptors, `filedes[0]` and `filedes[1]`. The file descriptors reference two data streams. Historically, pipes were unidirectional, and data flowed in one direction only. If two-way communication was needed, two pipes were opened: one for reading and another for writing. This is still true in Linux today. However, in some versions of UNIX (such as Solaris) the file descriptors returned by **pipe** are full duplex (bidirectional) and are both opened for reading/writing.

In a full duplex setting, if the process writes to `filedes[0]`, then `filedes[1]` is used for reading; otherwise, the process writes to `filedes[1]`, and `filedes[0]` is used for reading. In a half duplex setting (such as in Linux) `filedes[1]` is *always* used for writing, and `filedes[0]` is *always* used for reading—an attempt to **write** to `fildes[0]` or **read** from `filedes[1]` will produce an error (i.e., bad file descriptor).

If the **pipe** system call fails, it returns a –1 and sets `errno` (Table 5.6).

Table 5.6 `pipe` Error Messages.

#	Constant	`perror` Message	Explanation
23	ENFILE	File table overflow	System file table is full.
24	EMFILE	Too many open files	Process has exceeded the limit for number of open files.
14	EFAULT	Bad address	`filedes` is invalid.

As previously noted, data in a pipe is read on a FIFO basis. Program 5.1 shows a pair of processes (parent/child) that use a pipe to send the first

argument passed on the command line to the parent as a *message* to the child. Notice that the pipe is established prior to forking the child process.

Program 5.1 Parent/child processes communicating via a pipe.

```
File : p5.1.cxx
  |     /* Using a pipe to send data from a parent to a child process
  |      */
  |     #include <iostream>
  |     #include <cstdio>
  +     #include <unistd.h>
  |     #include <string.h>
  |     using namespace std;
  |     int
  |     main(int argc, char *argv[ ]) {
 10       int            f_des[2];
  |       static char    message[BUFSIZ];
  |       if (argc != 2) {
  |         cerr << "Usage: " << *argv << " message\n";
  |         return 1;
  +       }
  |       if (pipe(f_des) == -1) {             // generate the pipe
  |         perror("Pipe");     return 2;
  |       }
  |       switch (fork( )) {
 20       case -1:
  |         perror("Fork");     return 3;
  |       case 0:                              // In the child
  |         close(f_des[1]);
  |         if (read(f_des[0], message, BUFSIZ) != -1) {
  +           cout << "Message received by child: [" << message
  |                << "]" << endl;
  |           cout.flush();
  |         } else {
  |           perror("Read");    return 4;
 30         }
  |         break;
  |       default:                             // In the Parent
  |         close(f_des[0]);
  |         if (write(f_des[1], argv[1], strlen(argv[1])) != -1) {
  +           cout << "Message sent by parent   : [" <<
  |                argv[1] << "]" << endl;
  |           cout.flush();
  |         } else {
  |           perror("Write");   return 5;
 40         }
  |       }
  |       return 0;
  |     }
```

In the parent process the "read" pipe file descriptor `f_des[0]` is closed, and the message (the string referenced by `argv[1]`) is written to the pipe file descriptor `f_des[1]`. In the child process the "write" pipe file descriptor `f_des[1]` is closed, and pipe file descriptor `f_des[0]` is **`read`** to obtain the message. While the closing of the unused pipe file descriptors is not required, it is a good practice. Remember that for **`read`** to be successful, the number of bytes of data requested must be present in the pipe or all the **`write`** file descriptors for the pipe must be closed so that an end-of-file can be returned. The pipe file descriptors `f_des[0]` in the child and `f_des[1]` in the parent will be closed when each process exits. The output of Program 5.1 is shown in Figure 5.2.

Figure 5.2 Output of Program 5.1.

```
linux$ p5.1 Once_upon_a_starry_night
Message sent by parent   : [Once_upon_a_starry_night]
Message received by child: [Once_upon_a_starry_night]
```

5-1 EXERCISE

Modify Program 5.1 so the child, upon receipt of the message, changes its case and returns the message (via a pipe) to the parent, where it is then displayed. On a system that does not support duplex pipes, you will need to generate two pipes prior to forking the child process.

At a command-line level, a pipe is specified by the | symbol. As shown in Figure 5.3, pipes are used to tie the standard output of one command to the standard input of another to create a command pipeline.

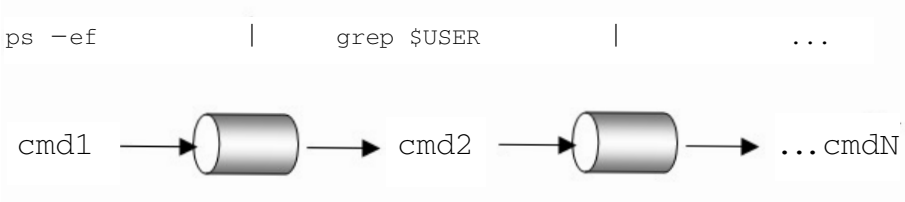

Figure 5.3 Using pipes on the command line.

For example, the command line sequence

```
linux$ ps –ef  | grep $USER | cat -n
```

will execute the **ps** `-ef` command (which displays, in full form, the process status of all users) and pipe its output to the **grep** `$USER` command. The **grep** command prints those lines that contain the contents of the variable `$USER`—that is, the user's login. A second pipe passes the output of the **grep** command to the **cat**, which (with option `–n`) displays its output as a numbered list. The redirection of the output of the **ps** command to be the input to the **grep** command and the output of the **grep** command to be the input of the **cat** command is accomplished with the inclusion of the command-line specification of a pipe. To achieve a similar arrangement with our parent/child pair, we need a way to associate standard input and standard output with the pipe we have created. This can be done either by using the **dup** or the **dup2** system call (Tables 5.7 and 5.8).

The **dup2** call supersedes the **dup** system call, but both bear discussion. The **dup** system call duplicates an original open file descriptor. The *new* descriptor references the system file table entry for the next available

Table 5.7 Summary of the `dup` System Call.

Include File(s)	`<unistd.h>`		Manual Section **2**
Summary	`int dup( int oldfd );`		
Return	Success	Failure	Sets `errno`
	Next available nonnegative file descriptor	–1	Yes

Table 5.8 Summary of the `dup2` System Call.

Include File(s)	`<unistd.h>`		Manual Section **2**
Summary	`int dup2( int oldfd, int newfd );`		
Return	Success	Failure	Sets `errno`
	`newfd` as a file descriptor for `oldfd`	–1	Yes

nonnegative file descriptor. The new descriptor will share the same file pointer (offset), have the same access mode as the original, and share locks. Both will remain open across an **exec** call, but they do not, however, share the close-on-exec flag. An important point to consider is that when called, **dup** will *always* return the next lowest available file descriptor.

A code sequence of

```
int f_des[2];
pipe(f_des);
close( fileno(stdout) );  // close standard output
dup(f_des[1]);            // duplicate 1st free descriptor
                             as write end of pipe
.
.
.
```

declares and generates a pipe. The file descriptor for standard output (say, file descriptor 1) is closed. The following **dup** system call returns the next lowest available file descriptor, which in this case should be the previously closed standard output file descriptor (i.e., 1). Thus, any data written to standard output in following statements would now be written to the pipe. Notice that there are two steps in this sequence: closing the descriptor and then **dup**-ing it. There is an outside chance that the sequence will be interrupted and the descriptor returned by **dup** will not be the one that was just closed. This could happen if a signal was caught and the signal-catching routine closed a file.

Enter the **dup2** system call. The **dup2** system call closes and duplicates the file descriptor as a single *atomic* action. When calling **dup2**, there is no time at which `newfd` is closed and `oldfd` has not yet been duplicated. If the file referenced by `newfd` is already open, it will be closed before the duplication is performed. For those more stout of heart, both the **dup** and **dup2** calls can be implemented with the **fcntl** system call (when passed the proper flag values).

A short program that mimics the `last | sort` command-line sequence is shown in Program 5.2. The files/pipes for the two processes, once Program 5.2 successfully executes the **fork** system call in line 17, are shown in Figure 5.4.

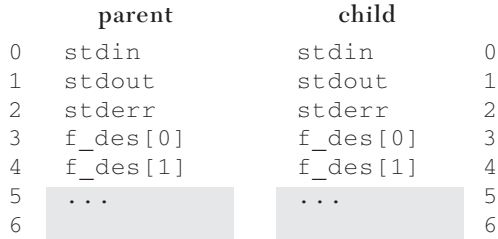

Figure 5.4 Initial entries for files/pipes.

Assuming a fairly standard setting (i.e., `stdin` = 0, `stdout` = 1, `stderr` = 2) with both `stdout` and `stderr` mapped to the same device (most likely the terminal), initially both the parent and child processes reference the same entries in the system file table. After the child process is generated, we use the **`dup2`** call to close standard output and duplicate it. The system returns the previous reference for standard output, which is now associated with the file table entry for `f_des[1]`. Once this association has been made, the file descriptors `f_des[0]` and `f_des[1]` are closed, as they are not needed by the child process.

Program 5.2 A `last` | `sort` pipeline.

```
File : p5.2.cxx
  |     /* A home grown last | sort cmd pipeline
  |      */
  |     #define_GNU_SOURCE
  |     #include <iostream>
  +     #include <cstdio>
  |     #include <unistd.h>
  |     using namespace std;
  |     enum { READ, WRITE };
  |
 10     int
  |     main( ) {
  |       int      f_des[2];
  |       if (pipe(f_des) == -1) {
  |         perror("Pipe");
  +         return 1;
  |       }
  |       switch (fork( )) {
  |       case -1:
  |         perror("Fork");
 20         return 2;
  |       case 0:                                  // In the child
  |         dup2( f_des[WRITE], fileno(stdout));
  |         close(f_des[READ] );
  |         close(f_des[WRITE]);
  +         execl("/usr/bin/last", "last", (char *) 0);
  |         return 3;
  |       default:                                   // In the parent
  |         dup2( f_des[READ], fileno(stdin));
  |         close(f_des[READ] );
 30         close(f_des[WRITE]);
  |         execl("/bin/sort", "sort", (char *) 0);
  |         return 4;
  |       }
  |       return 0;
  +     }
```

In the parent process the **dup2** call closes standard input and duplicates it as the reference `f_des[0]`. The entries for the files/pipes would now look like those shown in Figure 5.5. In the parent process, `stdout` and `stderr` have not been modified. However, `stdin` is now the read end of the pipe shared with the child. In the child process, `stdout` and `stderr` are their default values. However, `stdout` has been associated with the write end of pipe shared with the parent.

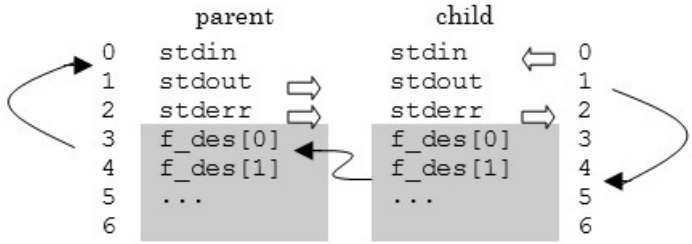

Figure 5.5 End entries for files/pipes.

When running Program 5.2, the two processes (parent and child) are running concurrently (at the same time). The sequence in which these processes will be executed is not guaranteed. For the processes involved, this is not a concern, since the pipe allows both processes to write/read at the same time.

We can summarize the steps involved for communication via unnamed pipes:

1. Create the pipe(s) needed.
2. Generate the child process(es).
3. Close/duplicate file descriptors to properly associate the *ends* of the pipe.
4. Close the unneeded ends of the pipe.
5. Perform the communication activities.
6. Close any remaining open file descriptors.
7. If appropriate, wait for child processes to terminate.

If either **dup** or **dup2** fail, they return a −1 and set `errno`. The error codes for **dup** and **dup2** are shown in Table 5.9.

Table 5.9 `dup/dup2` Error Messages.

#	Constant	`perror` Message	Explanation
4	EINTR	Interrupted system call	Signal was caught during the system call.
9	EBADF	Bad file descriptor	The file descriptor is invalid.
24	EMFILE	Too many open files	Process has exceeded the limit for number of open files.
67	ENOLINK	The link has been severed	The file descriptor value references a remote system that is no longer active.

5-2 EXERCISE

Most UNIX-based systems include a utility program called **`tee`** that copies standard input to standard output and to the file descriptor passed on the command line. Thus, the command sequence

```
linux$ cat x.c | tee /dev/tty | wc
```

would **`cat`** the contents of the file `x.c` and pipe the standard output to **`tee`**. The **`tee`** program would copy its standard input (from the **`cat`** command) to the file `/dev/tty` and to its standard output, where it would be piped to the **`wc`** (word count) program. Using unnamed pipes, write your own version of **`tee`** called **`my_tee`**. *Hint:* If you do not know your terminal device, on most systems the command **`stty`** will display the device. If **`stty`** does not work, try the **`who`** command. When passing the name of the terminal device to your **`my_tee`** program, be sure to include the full path for the device.

5-3 EXERCISE

Modify Program 5.2 so a variable number of commands can be passed to the program. Each command passed to the program should be *connected* to the next command via a pipe. When using this new program, a three-command sequence such as

```
linux$ last | sort | more
```

would be indicated as

```
linux$ my_p5.2  last  sort  more
```

5-4 EXERCISE

Rework the program written for Exercise 4.3 (the producer/consumer problem in Chapter 4) so the producer and consumer now use a pipe to communicate with one another.

Since the sequence of generating a pipe, forking a child process, duplicating file descriptors, and passing command execution information from one process to another via the pipe is relatively common, a set of standard library functions is available to simplify this task: **popen** and **pclose**. See Tables 5.10 and 5.11.

Table 5.10 Summary of the `popen` Library Function.

Include File(s)	`<stdio.h>`		Manual Section	**3**
Summary	`FILE *popen( const char *command, const, char *type )`			
Return	Success	Failure	Sets `errno`	
	Pointer to a FILE	NULL pointer	Sometimes	

Table 5.11 Summary of the `pclose` Library Function.

Include File(s)	`<stdio.h>`		Manual Section	**3**
Summary	`int pclose( FILE *stream );`			
Return	Success	Failure	Sets `errno`	
	Exit status of command	−1	Sometimes	

When successful, the **popen** call returns a pointer to a file stream (not an integer file descriptor). The arguments for **popen** are a pointer to the shell `command`[2] that will be executed and an I/O mode `type`. The I/O mode `type`

[2] This can be any valid Bourne shell command, including those with I/O redirection. Most often, the command is placed in a doubly quoted string.

(read or write) determines how the process will handle the file pointer returned by the **popen** call.

When invoked, the **popen** call automatically generates a child process. The child process **exec**s a Bourne shell (**/bin/sh**), which will execute the passed shell command. Input to and output from the child process is accomplished via a pipe. If the I/O mode `type` for **popen** is specified as `w` the parent process can **write** to the standard input of the shell command. In other terms, writing to the file pointer reference generated by the **popen** in the parent process will enable the child process running the shell command to **read** the data as its standard input. Conversely, if the I/O type is `r`, using the **popen** file pointer, the parent process can **read** from the standard output of the shell command (run by the child process). By default, the I/O stream generated by **popen** is fully buffered.

If **popen** fails due to an inability to allocate memory, `errno` will not be set. However, if the mode `type` is specified incorrectly, **popen** sets `errno` to EINVAL.

The **pclose** call is used to close a data stream opened with a **popen** call. If the data stream being closed is associated with a **popen**, **pclose** returns the exit status of the shell command referenced by the **popen**. If the data stream is not associated with a **popen** call, the **pclose** call returns a value of –1. If **pclose** is unable to obtain the status of the child process, `errno` is set to ECHILD.

Program 5.3 shows one way the **popen** and **pclose** calls can be used to pipe the output of one shell command to the input of another.

Program 5.3 Using `popen` and `pclose`.

```
File : p5.3.cxx
  |     /* Using the popen and pclose I/O commands
  |      */
  |     #define_GNU_SOURCE
  |     #include <iostream>
  +     #include <cstdio>
  |     #include <limits.h>
  |     #include <unistd.h>
  |     using namespace std;
  |     int
 10     main(int argc, char *argv[ ]) {
  |       FILE    *fin, *fout;
  |       char    buffer[PIPE_BUF];
  |       int     n;
  |       if (argc < 3) {
  +         cerr << "Usage " << argv << "cmd1 cmd2" << endl;
  |         return 1;
  |      }
```

```
  |       fin  = popen(argv[1], "r");
  |       fout = popen(argv[2], "w");
 20       fflush(fout);
  |       while ((n = read(fileno(fin), buffer, PIPE_BUF)) > 0)
  |         write(fileno(fout), buffer, n);
  |       pclose(fin);
  |       pclose(fout);
  +       return 0;
  |     }
```

As written, Program 5.3 requires two command-line arguments: two shell commands whose standard output/input is *redirected* via pipes generated when using the **popen** call. The first **popen** call, with the I/O option of `r`, directs the system to **fork** a child process that will execute the shell command referenced by `argv[1]`. The output of the command will be redirected so it can be **read** by the parent process when using the file pointer reference `fin`. In a similar manner, the second **popen**, with the I/O option of `w` directs the system to **fork** a second child process. As this child process executes its shell command (referenced by `argv[2]`), its standard input will be the data written to the pipe by the parent process, and its output will go the standard output. The parent process writes data to the second pipe using the file pointer reference `fout` and reads data from the first pipe using the file pointer reference `fin`. The **while** loop in the program is used to copy the data from the output end of one pipe to the input end of the other. The call to **fflush** in line 20 of the program is used to clear buffered output so that it will not be interleaved with data in the pipe.

Figure 5.6 depicts the arrangement when the shell command **last** and **more** are passed on the command line to Program 5.3.

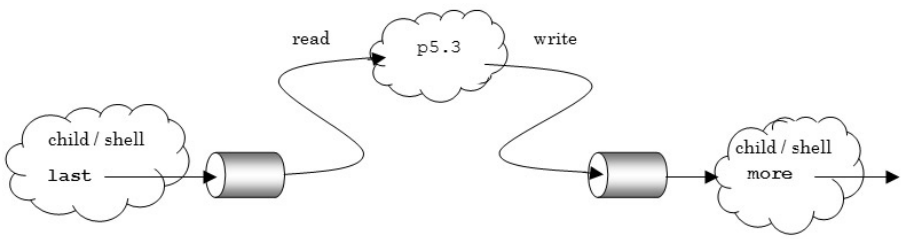

Figure 5.6 Program 5.3 relationships when invoked as `p5.3 last more`.

5-5 EXERCISE

Using *just* the **popen** call to generate pipes, can we create a pipeline consisting of *three* separate shell commands (e.g., a program that when passed three shell commands on the command line, would *pipe* the commands together in the manner `cmd1 | cmd2 | cmd3`)? If yes, write a program that shows how this can be done. If no, give the reason(s) why.

5.3 Named Pipes

UNIX provides for a second type of pipe called a named pipe, or FIFO (we will use the terms interchangeably). Named pipes are similar in spirit to unnamed pipes but have additional benefits. When created, named pipes have a directory entry. With the directory entry are file access permissions and the capability for unrelated processes to use the pipe file. Although the FIFO has a directory entry, keep in mind the data written to the FIFO is passed to and stored by the kernel and is not directly written to the file system.

Named pipes can be created at the shell level (on the command line) or within a program. It is instructive to look at the generation of a named pipe at the shell level before addressing its use in a program. At the shell level the command used to make a named pipe is **mknod**. Officially, **mknod** is a utility command designed to generate special files. It is most commonly used by the superuser to generate special device files (e.g., the block, character device files found in the `/dev` directory). For nonprivileged users, **mknod** can only be used to generate a named pipe. The syntax for the **mknod** command to make a named pipe is

```
linux$ mknod PIPE p
```

The first argument to the **mknod** command is the file name for the FIFO (this can be any valid UNIX file name; however, it is common to use an uppercase file name to alert the user to the *special* nature of the file). The second argument is a lowercase `p`, which notifies **mknod** that a FIFO file is to be created. If we issue the command shown above and check the directory entry for the file that it has created, we will find a listing similar to that shown below:

```
linux$ ls -l PIPE
prw-r--r--   1 gray     faculty         0 Feb 26 07:18 PIPE
```

The lowercase letter p at the start of the permission string indicates the file called PIPE is a FIFO. The default file permissions for a FIFO are assigned using the standard **umask** arrangement discussed previously. The number of bytes in the FIFO is listed as 0. As soon as all the processes that are using a named pipe are done with it, any remaining data in the pipe is released by the system and the byte count for the file reverts to 0. If we wish to, we can, on the command line, redirect the output from a shell command to a named pipe. If we do this, we should place the command sequence in the background to prevent it from hanging. We could then redirect the output of the same FIFO to be the input of another command.

For example, the command

```
linux$ cat test_file > PIPE &
[1] 27742
```

will cause the display of the contents of file test_file to be redirected to the named pipe PIPE. If this command is followed by

```
linux$ cat < PIPE
This is test
file to use
with our pipe.
[1]  + Done                          cat test_file > PIPE
```

the second **cat** command will read its input from the named pipe called PIPE and display its output to the screen.

5-6 EXERCISE

As long as there is one active reader and/or writer for a FIFO, the system will maintain its contents. Is it possible to produce a command sequence that proves this is so? Try issuing a command that generates a large amount of outpt (e.g., `cat p5.2.cxx`) and redirect the output to the FIFO, placing the command in the background. Follow this command with `ls -l` to see if the pipe actually has contents. Now try the following command sequence:

```
linux$ cat p5.2.cxx > PIPE & more < PIPE & ls -l PIPE
```

How do you explain the differences in output you observe?

While the previous discussion is instructive, it is of limited practical use. Under most circumstances, FIFOs are created in a programming environment, not on the command line. The system call to generate a FIFO in a program has the same name as the system command equivalent: **mknod** (Table 5.12).

Table 5.12 Summary of the **mknod** System Call.

Include File(s)	<sys/types.h> <sys/stat.h> <fcntl.h> <unistd.h>		Manual Section	**2**
Summary	int mknod(const char *pathname, mode_t mode, dev_t dev);			
Return	Success	Failure	Sets errno	
	0	−1	Yes	

The **mknod** system call creates the file referenced by pathname. The type of the file created (FIFO, character or block special, directory[3] or plain) and its access permissions are determined by the mode value. Most often the mode for the file is created by **OR**ing a symbolic constant indicating the file type with the file access permissions (see the section on **umask** for a more detailed discussion). Permissible file types are listed in Table 5.13.

Table 5.13 File Type Specification Constants for **mknod**.

Symbolic Constant	File Type
S_IFIFO	FIFO special
S_IFCHR	character special
S_IFDIR	directory
S_IFBLK	block special
S_IFREG	ordinary file

The dev argument for **mknod** is used only when a character or block special file is specified. For character and block special files, the dev argument is used to assign the major and minor number of the device. For nonprivileged users, the **mknod** system call can only be used to generate a FIFO. When generating a FIFO, the dev argument should be left as 0. If **mknod** is successful, it returns a value of 0. Otherwise, errno is set to indicate the error, and a value of –1 is returned.

[3] While most versions of **mknod** can also be used to generate a directory (if you are the superuser), the version found in Linux cannot (use the **mkdir** system call instead).

Table 5.14 `mknod` Error Messages.

#	Constant	`perror` Message	Explanation
1	EPERM	Operation not permitted	The effective ID of the calling process is not that of the superuser.
4	EINTR	Interrupted system call	Signal was caught during the system call.
12	ENOMEM	Cannot allocate memory	Insufficient kernel memory was available.
13	EACCES	Permission denied	Parent directory (or one of the directories in `pathname`) lacks write permission.
14	EFAULT	Bad address	`pathname` references an illegal address.
17	EEXIST	File exists	`pathname` already exists.
20	ENOTDIR	Not a directory	Part of the specified `pathname` is not a directory.
22	EINVAL	Invalid argument	Invalid `dev` specified.
28	ENOSPC	No space left on device	File system has no inodes left for new file generation.
30	EROFS	Read-only file system	Referenced file is (or would be) on a read-only file system.
67	ENOLINK	The link has been severed	The `pathname` value references a remote system that is no longer active.
72	EMULTIHOP	Multihop attempted	The `pathname` value requires multiple hops to remote systems, but file system does not allow it.
36	ENAMETOOLONG	File name too long	The `pathname` value exceeds system path/file name length.
40	ELOOP	Too many levels of symbolic links	The `perror` message says it all.

In many versions of UNIX, a C library function called **`mkfifo`** simplifies the generation of a FIFO. The **`mkfifo`** library function (Table 5.15) uses the **`mknod`** system call to generate the FIFO. Most often, unlike **`mknod`**, **`mkfifo`** does not require the user have superuser privileges.

Table 5.15 Summary of the `mkfifo` Library Function.

Include File(s)	`<sys/types.h>` `<sys/stat.h>`		Manual Section	**3**
Summary	`int mkfifo (const char *pathname, mode_t mode)`			
Return	Success	Failure	Sets `errno`	
	0	−1	Yes	

If **`mkfifo`** is used in place of **`mknod`**, the `mode` argument for **`mkfifo`** refers only to the file access permission for the FIFO, because the file type, by default, is set to S_IFIFO. If the **`mkfifo`** call fails, it returns a –1 and sets the value in `errno`. When generating a FIFO, the errors that may be encountered with **`mkfifo`** are similar to those previously listed for the **`mknod`** system call (Table 5.14). In our examples, we use the more universal **`mknod`** system call when generating a FIFO.

Our next example is somewhat more grand in scope than some of the past examples. We combine the use of unnamed and named pipes to produce a **client–server** relationship. Both the client and server processes will run on the same platform. The single-server process is run first and placed in the background. Client processes, run subsequently, are in the foreground. The client processes accept a shell command from the user. The command is sent to the server via a *public* FIFO (known to all clients and the server) for processing. Once the command is received, the server executes it using the **`popen–pclose`** sequence (which generates an unnamed pipe in the process). The server process returns the output of the command to the client over a *private* FIFO where the client, upon receipt, displays it to the screen. Figure 5.7 shows the process and pipe relationships.

More succinctly, the steps taken by the processes involved are as follows:

- Server generates the public FIFO (available to all participating client processes).
- Client process generates its own private FIFO.
- Client prompts for, and receives, a shell command.
- Client writes the name of its private FIFO and the shell command to the public FIFO.
- Server reads the public FIFO and obtains the private FIFO name and the shell command.

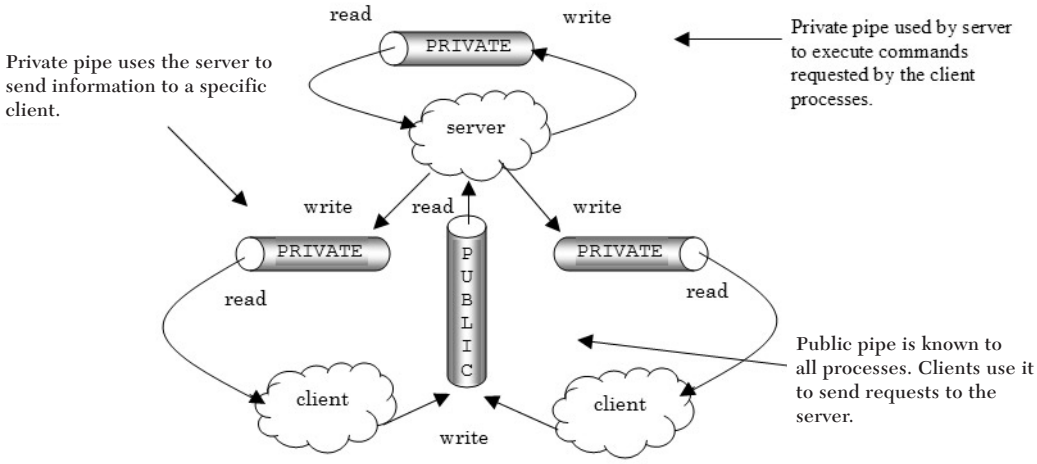

Figure 5.7 Client–server process relationships.

- Server uses a **`popen–pclose`** sequence to execute the shell command. The output of the shell command is sent back to the client via the private FIFO.
- Client displays the output of the command.

To ensure that both server and client processes will use the same public FIFO name and have the same message format, a local header file is used. This header file is shown in Figure 5.8.

Figure 5.8 Header file for client–server example.

```
File : local.h
  |     /*
  |        local header file for pipe client-server
  |      */
  |     #include <cstdio>
  +     #include <sys/types.h>
  |     #include <sys/stat.h>
  |     #include <fcntl.h>
  |     #include <unistd.h>
  |     #include <string.h>
 10     #include <linux/limits.h>
  |     #include <stdlib.h>
  |     using namespace std;
  |     const char *PUBLIC = "/tmp/PUBLIC";
  |     const int   B_SIZ  = (PIPE_BUF / 2);
  +     struct message {
```

Establish the name of the common public FIFO.

```
|       char   fifo_name[B_SIZ];
|       char   cmd_line[B_SIZ];
|     };
```

In this file, a constant is used to establish the name for the public FIFO as `/tmp/PUBLIC`. The format of the message that will be sent over the public FIFO is declared with the `struct` statement. The message structure consists of two character array members. The first member, called `fifo_name`, stores the name of the private FIFO. The second structure member, `cmd_line`, stores the command to be executed by the server.

Program 5.4 shows the code for the client process.

Program 5.4 The client process.

```
File : client.cxx
  |      /* The client process */
  |      #define _GNU_SOURCE
  |
  |      #include "local.h"
  +      int
  |      main(  ){
  |        int             n, privatefifo, publicfifo;
  |        static char     buffer[PIPE_BUF];
  |        struct message  msg;
 10
  |        sprintf(msg.fifo_name, "/tmp/fifo%d", getpid( ));
  |
  |        if (mknod(msg.fifo_name, S_IFIFO | 0666, 0) < 0) {
  |          perror(msg.fifo_name);
  +          return 1;
  |        }
  |        if ((publicfifo = open(PUBLIC, O_WRONLY)) == -1) {
  |          perror(PUBLIC);
  |          return 2;
 20        }
  |        while ( 1 ) {
  |          write(fileno(stdout), "\ncmd>", 6);
  |          memset(msg.cmd_line, 0x0, B_SIZ);
  |          n = read(fileno(stdin), msg.cmd_line, B_SIZ);
  +          if (!strncmp("quit", msg.cmd_line, n-1))
  |            break;
  |          write(publicfifo, (char *) &msg, sizeof(msg));
  |          if ((privatefifo = open(msg.fifo_name, O_RDONLY)) == -1) {
  |            perror(msg.fifo_name);
 30           return 3;
  |          }
```

Build a unique name for the *private* FIFO for this process.

Generate the *private* FIFO.

Open the *public* FIFO for writing.

Prompt for command; clear space to hold command.

Write command to public pipe for server to process.

```
|          while ((n = read(privatefifo, buffer, PIPE_BUF)) > 0) {
|            write(fileno(stderr), buffer, n);
|          }
+          close(privatefifo);
|        }
|        close(publicfifo);
|        unlink(msg.fifo_name);
|        return 0;
40     }
```

Open *private* FIFO; read what is returned.

Using the `sprintf` function, the client creates a unique name for its private FIFO by incorporating the value returned by the **`getpid`** system call. The **`mknod`** system call is used next to create the private FIFO with read and write permissions for all. The following **`open`** statement opens the public FIFO for writing. If for some reason the public FIFO has not been previously generated by the server, the **`open`** will fail. In this case the **`perror`** call produces an error message and the client process exits. If the **`open`** is successful, the client process then enters an endless loop. The client first prompts the user for a command.[4] Prior to obtaining the command, the structure member where the command will be stored is set to all NULLs using the C library function **`memset`**. This action assures that no extraneous characters will be left at this storage location. Note that using **`memset`** is preferable to using the deprecated **`bzero`** library function for clearing a string. The **`read`** statement in line 24 obtains the user's input from standard input and stores it in `msg.cmd_line`. The input is checked to determine if the user would like to quit the program. The check is accomplished by comparing the input to the character string `quit`. We use `n-1` as the number of characters for comparison to avoid including the `\n` found at the end of the user's input. If `quit` was entered, the `while` loop is exited via the `break` statement, the private FIFO is removed, and the client process terminates. If the user does not want to quit, the entire message structure, consisting of the private FIFO name and the command the user entered, is written to the public FIFO (thus sending the information on to the server). The client process then attempts to **`read`** its private FIFO to obtain the output that will be sent to it from the server. At this juncture, if the server has not finished with its execution of the client's command, the client process will block (which is the default for **`read`**). Once data is available from the private FIFO, the `while` loop in the client will **`read`** and **`write`** its contents to standard error. The code for the server process is shown in Program 5.5.

[4] Notice that all the I/O in the program is done with **`read/write`** to avoid buffer flushing problems associated with standard I/O library calls.

Program 5.5 The server process.

```
File : server.cxx
  |     /* The server process */
  |     #define _GNU_SOURCE
  |
  |     #include "local.h"
  +     int
  |     main( ){
  |       int             n, done, dummyfifo, publicfifo, privatefifo;
  |       struct message  msg;
  |       FILE            *fin;
 10       static char     buffer[PIPE_BUF];
  |
  |       mknod(PUBLIC, S_IFIFO | 0666, 0);
  |
  |       if ((publicfifo = open(PUBLIC, O_RDONLY)) == -1 ||
  +           (dummyfifo  = open(PUBLIC, O_WRONLY | O_NDELAY)) == -1 ) {
  |           perror(PUBLIC);
  |           return 1;
  |       }
  |
 20       while (read(publicfifo, (char *) &msg, sizeof(msg)) > 0) {
  |         n = done = 0;
  |         do {
  |           if ((privatefifo=open(msg.fifo_name,
  |                                 O_WRONLY|O_NDELAY)) == -1)
  +             sleep(3);
  |           else {
  |             fin = popen(msg.cmd_line, "r");
  |             write(privatefifo, "\n", 1);
  |             while ((n = read(fileno(fin), buffer, PIPE_BUF)) > 0) {
 30               write(privatefifo, buffer, n);
  |                memset(buffer, 0x0, PIPE_BUF);
  |             }
  |             pclose(fin);
  |             close(privatefifo);
  +             done = 1;
  |           }
  |         } while (++n < 5 && !done);
  |         if (!done) {
  |           write(fileno(stderr),
 40          "\nNOTE: SERVER ** NEVER ** accessed private FIFO\n", 48);
  |           return 2;
  |         }
  |       }
  |       return 0;
  +     }
```

Generate *public* FIFO and open for reading and writing.

Read message (command) from *public* FIFO.

Open the child's *private* FIFO.

Server executes the command using **popen**.

Command output is read and sent to the child.

The server process is responsible for creating the public FIFO. Once created, the public FIFO is opened for both reading and writing. This may appear to be a little odd, as the server process only needs to **read** from the public FIFO. By opening the public FIFO for writing as well, the public FIFO always has at least one writing process associated with it. Therefore, the server process will never receive an end-of-file on the public FIFO. The server process will block on an empty public FIFO waiting for additional messages to be written. This technique saves us from having to close and reopen the public FIFO every time a client process finishes its activities.

Once the public FIFO is established, the server attempts to **read** a message from the public FIFO. When a message is **read** (consisting of a private FIFO name and a command to execute), the server tries to **open** the indicated private FIFO for writing. The attempt to **open** the private FIFO is done within a `do-while` loop. The O_NDELAY flag is used to keep the **open** from generating a deadlock situation. Should the client, for some reason, not open its end of the private FIFO for reading, the server would, without the O_NDELAY flag specification, block at the **open** of the private FIFO for writing. If the attempt to open the private FIFO fails, the server sleeps three seconds and tries again. After five unsuccessful attempts, the server displays an informational message to standard error and continues with its processing. If the private FIFO is successfully opened, a **popen** is used to execute the command that was passed in the message structure. The output of the command (which is obtained from the unnamed pipe) is written to the private FIFO using a `while` loop. When all of the output of the command has been written to the unnamed pipe, the unnamed pipe and private FIFO are closed. A sample run of the client-server programs is shown in Figure 5.9.

Figure 5.9 Typical client-server output.

```
linux$ server &                    <-- Place the server in the background.
[1] 27107
$ client                           <-- Run a client process in the foreground.
cmd>ps
  PID TTY          TIME CMD
14736 pts/3    00:00:00 csh
27107 pts/3    00:00:00 server
27108 pts/3    00:00:00 client
27109 pts/3    00:00:00 6
cmd>who
gray     pts/3    Feb 27 11:28
cmd>quit                           <-- The server process must be removed by
linux$ kill -9 27107                   sending it a kill signal.
[1]    Killed              server
$
```

The server process is placed in the background. The client process is then run, and shell commands (**ps** and **who**) are entered in response to the `cmd>` prompt. The output of each command (after it is executed by the server process and its output sent back to the client) is shown. The client process is terminated by entering the word `quit`. The server process, which remains in the background even after the client has been removed, is terminated by using the **kill** command.

5-7 EXERCISE

There are a number of additions that can be made to the client program to make it more robust. For example, if the client exits due to the receipt of an interrupt signal (CTRL+C), the private FIFO is not removed. Use a signal-catching routine to correct this oversight. When the client process is initiated, it will fail if the server process is not available. Correct this by having the client start the server process if it is not active.

5-8 EXERCISE

As written, the server program will process each command request in turn. Should one of these requests require a long time to execute, all other client processes must wait to be serviced. Rewrite the server program so that when the server process receives a message, it **fork**s a child process to carry out the task of executing the command and returning the output of the command to the client process.

5.4 Summary

Pipes provide the user with a more reliable, synchronized means of interprocess communication. Unnamed pipes can be used only with related processes. The **popen** system call provides the user with an easy way to generate an unnamed pipe to execute a shell command. Named pipes (FIFOs), which exist as actual directory entries, can be shared by unrelated processes. The

amount of data a pipe can contain is limited by the system. When a pipe is no longer associated with any processes, its contents are flushed by the system. The **read** and **write** system calls, which can be used with pipes, provide the user with an easy means of coordinating the flow of data in a pipe. Care must be taken when using pipes to prevent deadlock situations. Deadlock can occur when one process opens one end of a pipe for writing and another process opens the other end of the same pipe for writing. Each process in turn is waiting for the other to complete its action.[5] Pipes can be used only by processes that are running on the same platform. Unfortunately, pipes provide no easy way for a reading process to determine who the writing process was. All processes involved with using pipes must have forehand knowledge of their existence.

5.5 Key Terms and Concepts

bzero library function
dup system call
dup2 system call
FIFO
memset library function
mkdir library function
mkfifo library function
mknod command
mknod system call
named pipe
O_NDELAY flag
O_NOBLOCK flag
pclose library function
pipe
pipe system call
PIPE_BUF
PIPE_SIZE
popen I/O function
private FIFO
public FIFO
read system call
tee command
unnamed pipe
write system call

[5] As the unnamed pipe generated by **popen** is done without the user's direct use of the **open** system call, should the O_NDELAY or O_NBLOCK flags need to be set, the **fcntl** system call must be used.