Wrox Programmer to Programmer™

# Beginning Linux® Programming

4th Edition

Neil Matthew, Richard Stones

wrox

*Updates, source code, and Wrox technical support at* **www.wrox.com**

# 13

# Inter-Process Communication: Pipes

In Chapter 11, you saw a very simple way of sending messages between two processes using signals. You created notification events that could be used to provoke a response, but the information transferred was limited to a signal number.

In this chapter, you take a look at pipes, which allow more useful data to be exchanged between processes. By the end of the chapter, you'll be using your newfound knowledge to re-implement the CD database program as a very simple client/server application.

We cover the following topics in this chapter:

- The definition of a pipe
- Process pipes
- Pipe calls
- Parent and child processes
- Named pipes: FIFOs
- Client/server considerations

## What Is a Pipe?

We use the term *pipe* to mean connecting a data flow from one process to another. Generally you attach, or pipe, the output of one process to the input of another.

Most Linux users will already be familiar with the idea of a pipeline, linking shell commands together so that the output of one process is fed straight to the input of another. For shell commands, this is done using the pipe character to join the commands, such as

```
cmd1 | cmd2
```

The shell arranges the standard input and output of the two commands, so that

- ❑ The standard input to `cmd1` comes from the terminal keyboard.
- ❑ The standard output from `cmd1` is fed to `cmd2` as its standard input.
- ❑ The standard output from `cmd2` is connected to the terminal screen.

What the shell has done, in effect, is reconnect the standard input and output streams so that data flows from the keyboard input through the two commands and is then output to the screen. See Figure 13-1 for a visual representation of this process.

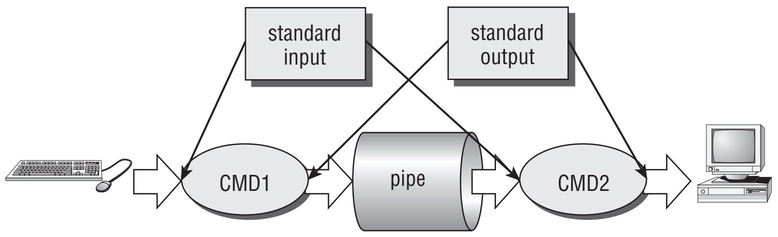

**Figure 13-1**

In this chapter, you see how to achieve this effect within a program and how you can use pipes to connect multiple processes to allow you to implement a simple client/server system.

## Process Pipes

Perhaps the simplest way of passing data between two programs is with the `popen` and `pclose` functions. These have the following prototypes:

```
#include <stdio.h>

FILE *popen(const char *command, const char *open_mode);
int pclose(FILE *stream_to_close);
```

### *popen*

The `popen` function allows a program to invoke another program as a new process and either pass data to it or receive data from it. The `command` string is the name of the program to run, together with any parameters. `open_mode` must be either `"r"` or `"w"`.

If the `open_mode` is `"r"`, output from the invoked program is made available to the invoking program and can be read from the file stream `FILE *` returned by `popen`, using the usual `stdio` library functions for reading (for example, `fread`). However, if `open_mode` is `"w"`, the program can send data to the invoked command with calls to `fwrite`. The invoked program can then read the data on its standard input. Normally, the program being invoked won't be aware that it's reading data from another process; it simply reads its standard input stream and acts on it.

A call to `popen` must specify either `"r"` or `"w"`; no other option is supported in a standard implementation of `popen`. This means that you can't invoke another program and both read from and write to it. On failure, `popen` returns a null pointer. If you want bidirectional communication using pipes, the normal solution is to use two pipes, one for data flow in each direction.

## *pclose*

When the process started with `popen` has finished, you can close the file stream associated with it using `pclose`. The `pclose` call will return only when the process started with `popen` finishes. If it's still running when `pclose` is called, the `pclose` call will wait for the process to finish.

The `pclose` call normally returns the exit code of the process whose file stream it is closing. If the invoking process has already executed a `wait` statement before calling `pclose`, the exit status will be lost because the invoked process has finished and `pclose` will return –1, with `errno` set to `ECHILD`.

## Try It Out Reading Output from an External Program

Let's try a simple `popen` and `pclose` example, `popen1.c`. You'll use `popen` in a program to access information from `uname`. The `uname -a` command prints system information, including the machine type, the OS name, version and release, and the machine's network name.

Having initialized the program, you open the pipe to `uname`, making it readable and setting `read_fp` to point to the output. At the end, the pipe pointed to by `read_fp` is closed.

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

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;
    memset(buffer, '\0', sizeof(buffer));
    read_fp = popen("uname -a", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        if (chars_read > 0) {
            printf("Output was:-\n%s\n", buffer);
        }
        pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

When you run this program, you should get output like the following (from one of the authors' machines):

```
$ ./popen1
Output was:-
Linux suse103 2.6.20.2-2-default #1 SMP Fri Mar 9 21:54:10 UTC 2007 i686 i686 i386
GNU/Linux
```

### How It Works

The program uses the popen call to invoke the uname command with the -a parameter. It then uses the returned file stream to read data up to BUFSIZ characters (as this is a #define from stdio.h) and then prints it out so it appears on the screen. Because you've captured the output of uname inside a program, it's available for processing.

## Sending Output to popen

Now that you've seen an example of capturing output from an external program, let's look at sending output to an external program. Here's a program, popen2.c, that pipes data to another. Here, you'll use od (octal dump).

### Try It Out Sending Output to an External Program

Have a look at the following code; you can see that it is very similar to the preceding example, except you are writing down a pipe instead of reading from it. This is popen2.c.

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

int main()
{
    FILE *write_fp;
    char buffer[BUFSIZ + 1];

    sprintf(buffer, "Once upon a time, there was...\n");
    write_fp = popen("od -c", "w");
    if (write_fp != NULL) {
        fwrite(buffer, sizeof(char), strlen(buffer), write_fp);
        pclose(write_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

When you run this program, you should get the following output:

```
$ ./popen2
0000000   O   n   c   e       u   p   o   n       a       t   i   m   e
0000020   ,       t   h   e   r   e       w   a   s   .   .   .  \n
0000037
```

### How It Works

The program uses popen with the parameter "w" to start the od -c command, so that it can send data to that command. It then sends a string that the od -c command receives and processes; the od -c command then prints the result of the processing on its standard output.

From the command line, you can get the same output with the command

```
$ echo "Once upon a time, there was..." | od -c
```

## Passing More Data

The mechanism that you've used so far simply sends or receives all the data in a single `fread` or `fwrite`. Sometimes you may want to send the data in smaller pieces, or perhaps you may not know the size of the output. To avoid having to declare a very large buffer, you can just use multiple `fread` or `fwrite` calls and process the data in parts.

Here's a program, `popen3.c`, that reads all of the data from a pipe.

### Try It Out Reading Larger Amounts of Data from a Pipe

In this program, you read data from an invoked `ps ax` process. There's no way to know in advance how much output there will be, so you must allow for multiple reads of the pipe.

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

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;

    memset(buffer, '\0', sizeof(buffer));
    read_fp = popen("ps ax", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        while (chars_read > 0) {
            buffer[chars_read - 1] = '\0';
            printf("Reading %d:-\n %s\n", BUFSIZ, buffer);
            chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        }
        pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

The output, edited for brevity, is similar to this:

```
$ ./popen3
Reading 1024:-
   PID TTY STAT  TIME COMMAND
    1  ?  Ss    0:03 init [5]
```

```
      2  ?   SW    0:00 [kflushd]
      3  ?   SW    0:00 [kpiod]
      4  ?   SW    0:00 [kswapd]
      5  ?   SW<   0:00 [mdrecoveryd]
...
    240 tty2 S     0:02 emacs draft1.txt
Reading 1024:-
    368 tty1 S     0:00 ./popen3
    369 tty1 R     0:00 ps -ax
...
```

### How It Works

The program uses popen with an "r" parameter in a similar fashion to popen1.c. This time, it continues reading from the file stream until there is no more data available. Notice that, although the ps command takes some time to execute, Linux arranges the process scheduling so that both programs run when they can. If the reader process, popen3, has no input data, it's suspended until some becomes available. If the writer process, ps, produces more output than can be buffered, it's suspended until the reader has consumed some of the data.

In this example, you may not see Reading:- output a second time. This will be the case if BUFSIZ is greater than the length of the ps command output. Some (mostly more recent) Linux systems set BUFSIZ as high as 8,192 or even higher. To test that the program works correctly when reading several chunks of output, try reading less than BUFSIZ, maybe BUFSIZE/10, characters at a time.

## How popen Is Implemented

The popen call runs the program you requested by first invoking the shell, sh, passing it the command string as an argument. This has two effects, one good and the other not so good.

In Linux (as in all UNIX-like systems), all parameter expansion is done by the shell, so invoking the shell to parse the command string before the program is invoked allows any shell expansion, such as determining what files *.c actually refers to, to be done before the program starts. This is often quite useful, and it allows complex shell commands to be started with popen. Other process creation functions, such as execl, can be much more complex to invoke, because the calling process has to perform its own shell expansion.

The unfortunate effect of using the shell is that for every call to popen, a shell is invoked along with the requested program. Each call to popen then results in two extra processes being started, which makes the popen function a little expensive in terms of system resources and invocation of the target command is slower than it might otherwise have been.

Here's a program, popen4.c, that you can use to demonstrate the behavior of popen. You can count the lines in all the popen example source files by cating the files and then piping the output to wc -l, which counts the number of lines. On the command line, the equivalent command is

```
$ cat popen*.c | wc -l
```

*Actually,* wc -l popen*.c *is easier to type and much more efficient, but the example serves to illustrate the principle.*

## Try It Out popen Starts a Shell

This program uses exactly the preceding command, but through `popen` so that it can read the result:

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

int main()
{
    FILE *read_fp;
    char buffer[BUFSIZ + 1];
    int chars_read;


    memset(buffer, '\0', sizeof(buffer));
    read_fp = popen("cat popen*.c | wc -l", "r");
    if (read_fp != NULL) {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        while (chars_read > 0) {
            buffer[chars_read - 1] = '\0';
            printf("Reading:-\n %s\n", buffer);
            chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        }
        pclose(read_fp);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

When you run this program, the output is

```
$ ./popen4
Reading:-
      94
```

### How It Works

The program shows that the shell is being invoked to expand `popen*.c` to the list of all files starting `popen` and ending in `.c` and also to process the pipe (`|`) symbol and feed the output from `cat` into `wc`. You invoke the shell, the `cat` program, and `wc` and cause an output redirection, all in a single `popen` call. The program that invokes the command sees only the final output.

# The Pipe Call

You've seen the high-level `popen` function, but now let's move on to look at the lower-level `pipe` function. This function provides a means of passing data between two programs, without the overhead of invoking a shell to interpret the requested command. It also gives you more control over the reading and writing of data.

The pipe function has the following prototype:

```
#include <unistd.h>

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

pipe is passed (a pointer to) an array of two integer file descriptors. It fills the array with two new file descriptors and returns a zero. On failure, it returns -1 and sets errno to indicate the reason for failure. Errors defined in the Linux manual page for pipe (in section 2 of the manual) are

- EMFILE: Too many file descriptors are in use by the process.
- ENFILE: The system file table is full.
- EFAULT: The file descriptor is not valid.

The two file descriptors returned are connected in a special way. Any data written to file_descriptor[1] can be read back from file_descriptor[0]. The data is processed in a *first in, first out* basis, usually abbreviated to *FIFO*. This means that if you write the bytes 1, 2, 3 to file_descriptor[1], reading from file_descriptor[0] will produce 1, 2, 3. This is different from a stack, which operates on a *last in, first out* basis, usually abbreviated to *LIFO*.

*It's important to realize that these are file descriptors, not file streams, so you must use the lower-level* read *and* write *system calls to access the data, rather than the stream library functions* fread *and* fwrite*.*

Here's a program, pipe1.c, that uses pipe to create a pipe.

## Try It Out The pipe Function

The following example is pipe1.c. Note the file_pipes array, the address of which is passed to the pipe function as a parameter.

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

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        data_processed = write(file_pipes[1], some_data, strlen(some_data));
        printf("Wrote %d bytes\n", data_processed);
        data_processed = read(file_pipes[0], buffer, BUFSIZ);
        printf("Read %d bytes: %s\n", data_processed, buffer);
```

```
            exit(EXIT_SUCCESS);
        }
        exit(EXIT_FAILURE);
    }
```

When you run this program, the output is

```
$ ./pipe1
Wrote 3 bytes
Read 3 bytes: 123
```

## How It Works

The program creates a `pipe` using the two file descriptors in the array `file_pipes[]`. It then writes data into the pipe using the file descriptor `file_pipes[1]` and reads it back from `file_pipes[0]`. Notice that the pipe has some internal buffering that stores the data in between the calls to `write` and `read`.

You should be aware that the effect of trying to write using `file_descriptor[0]`, or read using `file_descriptor[1]`, is undefined, so the behavior could be very strange and may change without warning. On the authors' systems, such calls fail with a –1 return value, which at least ensures that it's easy to catch this mistake.

At first glance, this example of a pipe doesn't seem to offer us anything that we couldn't have done with a simple file. The real advantage of pipes comes when you want to pass data between two processes. As you saw in Chapter 12, when a program creates a new process using the `fork` call, file descriptors that were previously open remain open. By creating a pipe in the original process and then `fork`ing to create a new process, you can pass data from one process to the other down the pipe.

## Try It Out Pipes across a fork

1. This is `pipe2.c`. It starts rather like the first example, up until you make the call to `fork`.

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

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    pid_t fork_result;

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == -1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }
```

2. You've made sure the `fork` worked, so if `fork_result` equals zero, you're in the child process:

```
        if (fork_result == 0) {
            data_processed = read(file_pipes[0], buffer, BUFSIZ);
            printf("Read %d bytes: %s\n", data_processed, buffer);
            exit(EXIT_SUCCESS);
        }
```

3. Otherwise, you must be in the parent process:

```
        else {
            data_processed = write(file_pipes[1], some_data,
                                   strlen(some_data));
            printf("Wrote %d bytes\n", data_processed);
        }
    }
    exit(EXIT_SUCCESS);
}
```

When you run this program, the output is, as before,

```
$ ./pipe2
Wrote 3 bytes
Read 3 bytes: 123
```

You may find that in practice the command prompt reappears before the last part of the output because the parent will finish before the child, so we have tidied the output here to make it easier to read.

## How It Works

First, the program creates a pipe with the `pipe` call. It then uses the `fork` call to create a new process. If the `fork` was successful, the parent writes data into the pipe, while the child reads data from the pipe. Both parent and child exit after a single `write` and `read`. If the parent exits before the child, you might see the shell prompt between the two outputs.

Although the program is superficially very similar to the first `pipe` example, we've taken a big step forward by being able to use separate processes for the reading and writing, as illustrated in Figure 13-2.

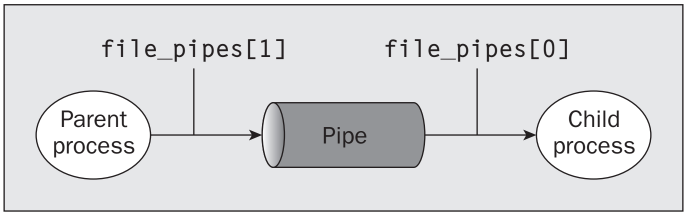

**Figure 13-2**

## Parent and Child Processes

The next logical step in our investigation of the pipe call is to allow the child process to be a different program from its parent, rather than just a different process running the same program. You do this using the exec call. One difficulty is that the new execed process needs to know which file descriptor to access. In the previous example, this wasn't a problem because the child had access to its copy of the file_pipes data. After an exec call, this will no longer be the case, because the old process has been replaced by the new child process. You can get around this by passing the file descriptor (which is, after all, just a number) as a parameter to the newly execed program.

To show how this works, you need two programs. The first is the *data producer*. It creates the pipe and then invokes the child, the *data consumer*.

### Try It Out Pipes and exec

1. For the first program, you adapt pipe2.c to pipe3.c. The changed lines are shown shaded:

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

int main()
{

    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    pid_t fork_result;

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == (pid_t)-1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }

        if (fork_result == 0) {
            sprintf(buffer, "%d", file_pipes[0]);
            (void)execl("pipe4", "pipe4", buffer, (char *)0);
            exit(EXIT_FAILURE);
        }
        else {
            data_processed = write(file_pipes[1], some_data,
                                   strlen(some_data));
            printf("%d - wrote %d bytes\n", getpid(), data_processed);
        }
    }
    exit(EXIT_SUCCESS);
}
```

2. The consumer program, `pipe4.c`, which reads the data, is much simpler:

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

int main(int argc, char *argv[])
{
    int data_processed;
    char buffer[BUFSIZ + 1];
    int file_descriptor;

    memset(buffer, '\0', sizeof(buffer));
    sscanf(argv[1], "%d", &file_descriptor);
    data_processed = read(file_descriptor, buffer, BUFSIZ);

    printf("%d - read %d bytes: %s\n", getpid(), data_processed, buffer);
    exit(EXIT_SUCCESS);
}
```

Remembering that `pipe3` invokes the `pipe4` program, you get something similar to the following output when you run `pipe3`:

```
$ ./pipe3
22460 - wrote 3 bytes
22461 - read 3 bytes: 123
```

### *How It Works*

The `pipe3` program starts like the previous example, using the `pipe` call to create a pipe and then using the `fork` call to create a new process. It then uses `sprintf` to store the "read" file descriptor number of the pipe in a buffer that will form an argument of `pipe4`.

A call to `execl` is used to invoke the `pipe4` program. The arguments to `execl` are

- The program to invoke
- `argv[0]`, which takes the program name
- `argv[1]`, which contains the file descriptor number you want the program to read from
- `(char *)0`, which terminates the parameters

The `pipe4` program extracts the file descriptor number from the argument string and then reads from that file descriptor to obtain the data.

## *Reading Closed Pipes*

Before we move on, we need to look a little more carefully at the file descriptors that are open. Up to this point you have allowed the reading process simply to read some data and then exit, assuming that Linux will clean up the files as part of the process termination.

Most programs that read data from the standard input do so differently than the examples you've seen so far. They don't usually know how much data they have to read, so they will normally loop — reading data, processing it, and then reading more data until there's no more data to read.

A `read` call will normally block; that is, it will cause the process to wait until data becomes available. If the other end of the pipe has been closed, then no process has the pipe open for writing, and the `read` blocks. Because this isn't very helpful, a `read` on a pipe that isn't open for writing returns zero rather than blocking. This allows the reading process to detect the pipe equivalent of end of file and act appropriately. Notice that this isn't the same as reading an invalid file descriptor, which `read` considers an error and indicates by returning –1.

If you use a pipe across a `fork` call, there are two different file descriptors that you can use to write to the pipe: one in the parent and one in the child. You must close the `write` file descriptors of the pipe in both parent and child processes before the pipe is considered closed and a `read` call on the pipe will fail. You'll see an example of this later when we return to this subject in more detail to look at the `O_NONBLOCK` flag and FIFOs.

## Pipes Used as Standard Input and Output

Now that you know how to make a `read` on an empty pipe fail, you can look at a much cleaner method of connecting two processes with a pipe. You arrange for one of the pipe file descriptors to have a known value, usually the standard input, 0, or the standard output, 1. This is slightly more complex to set up in the parent, but it allows the child program to be much simpler.

The one big advantage is that you can invoke standard programs, ones that don't expect a file descriptor as a parameter. In order to do this, you need to use the `dup` function, which you met in Chapter 3. There are two closely related versions of `dup` that have the following prototypes:

```
#include <unistd.h>

int dup(int file_descriptor);
int dup2(int file_descriptor_one, int file_descriptor_two);
```

The purpose of the `dup` call is to open a new file descriptor, a little like the `open` call. The difference is that the new file descriptor created by `dup` refers to the same file (or pipe) as an existing file descriptor. In the case of `dup`, the new file descriptor is always the lowest number available, and in the case of `dup2` it's the same as, or the first available descriptor greater than, the parameter `file_descriptor_two`.

*You can get the same effect as* `dup` *and* `dup2` *by using the more general* `fcntl` *call, with a command* `F_DUPFD`*. Having said that, the* `dup` *call is easier to use because it's tailored specifically to the needs of creating duplicate file descriptors. It's also very commonly used, so you'll find it more frequently in existing programs than* `fcntl` *and* `F_DUPFD`*.*

So how does `dup` help in passing data between processes? The trick is knowing that the standard input file descriptor is always 0 and that `dup` always returns a new file descriptor using the lowest available number. By first closing file descriptor 0 and then calling `dup`, the new file descriptor will have the number 0. Because the new descriptor is a duplicate of an existing one, standard input will have been changed to access the file or pipe whose file descriptor you passed to `dup`. You will have created two file descriptors that refer to the same file or pipe, and one of them will be the standard input.

## *File Descriptor Manipulation by close and dup*

The easiest way to understand what happens when you close file descriptor 0, and then call `dup`, is to look at how the state of the first four file descriptors changes during the sequence. This is shown in the following table.

| File Descriptor Number | Initially | After close of File Descriptor 0 | After dup |
|---|---|---|---|
| 0 | Standard input | {closed} | Pipe file descriptor |
| 1 | Standard output | Standard output | Standard output |
| 2 | Standard error | Standard error | Standard error |
| 3 | Pipe file descriptor | Pipe file descriptor | Pipe file descriptor |

### Try It Out Pipes and dup

Let's return to the previous example, but this time you'll arrange for the child program to have its `stdin` file descriptor replaced with the `read` end of the pipe you create. You'll also do some tidying up of file descriptors so the child program can correctly detect the end of the data in the pipe. As usual, we'll omit some error checking for the sake of brevity.

Modify `pipe3.c` to `pipe5.c` using the following code:

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

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    pid_t fork_result;

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == (pid_t)-1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }

        if (fork_result == (pid_t)0) {
            close(0);
            dup(file_pipes[0]);
            close(file_pipes[0]);
            close(file_pipes[1]);
```

```
                execlp("od", "od", "-c", (char *)0);
                exit(EXIT_FAILURE);
            }
            else {
                close(file_pipes[0]);
                data_processed = write(file_pipes[1], some_data,
                                       strlen(some_data));
                close(file_pipes[1]);
                printf("%d - wrote %d bytes\n", (int)getpid(), data_processed);
            }
        }
        exit(EXIT_SUCCESS);
    }
```

The output from this program is

```
$ ./pipe5
22495 - wrote 3 bytes
0000000   1   2   3
0000003
```

## How It Works

As before, the program creates a pipe and then forks, creating a child process. At this point, both the parent and child have file descriptors that access the pipe, one each for reading and writing, so there are four open file descriptors in total.

Let's look at the child process first. The child closes its standard input with `close(0)` and then calls `dup(file_pipes[0])`. This duplicates the file descriptor associated with the `read` end of the pipe as file descriptor 0, the standard input. The child then closes the original file descriptor for reading from the pipe, `file_pipes[0]`. Because the child will never write to the pipe, it also closes the `write` file descriptor associated with the pipe, `file_pipes[1]`. It now has a single file descriptor associated with the pipe: file descriptor 0, its standard input.

The child can then use `exec` to invoke any program that reads standard input. In this case, you use the `od` command. The `od` command will wait for data to be available to it as if it were waiting for input from a user terminal. In fact, without some special code to explicitly detect the difference, it won't know that the input is from a pipe rather than a terminal.

The parent starts by closing the read end of the pipe `file_pipes[0]`, because it will never read the pipe. It then writes data to the pipe. When all the data has been written, the parent closes the write end of the pipe and exits. Because there are now no file descriptors open that could write to the pipe, the `od` program will be able to read the three bytes written to the pipe, but subsequent reads will then return 0 bytes, indicating an end of file. When the read returns 0, the `od` program exits. This is analogous to running the `od` command on a terminal, then pressing Ctrl+D to send end of file to the `od` command.

Figure 13-3 shows the sequence after the call to the `pipe`, Figure 13-4 shows the sequence after the call to `fork`, and Figure 13-5 represents the program when it's ready to transfer data.

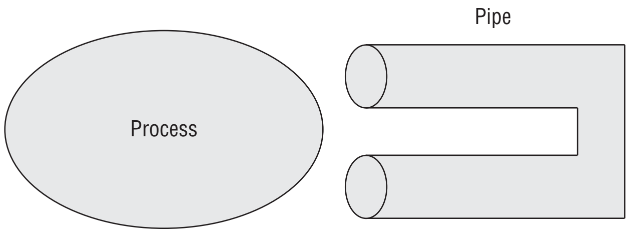

Figure 13-3

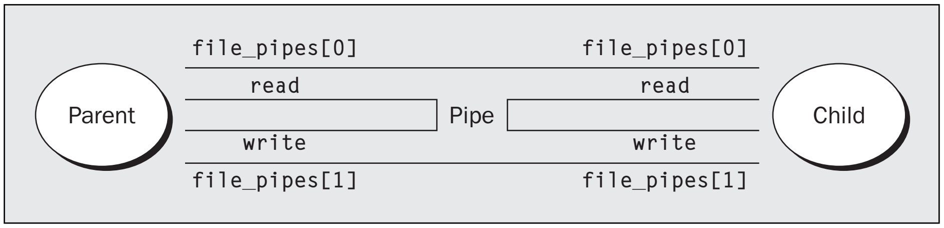

Figure 13-4

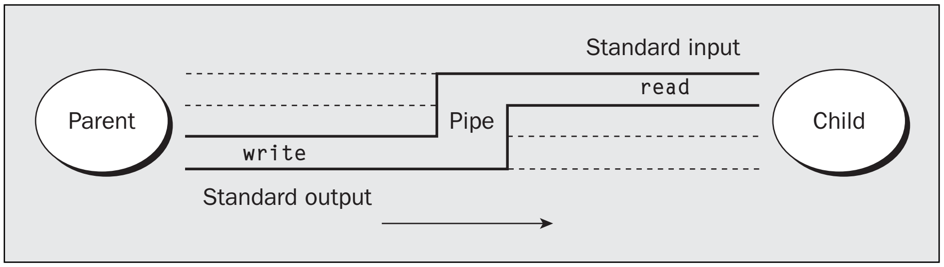

Figure 13-5

## Named Pipes: FIFOs

So far, you have only been able to pass data between related programs, that is, programs that have been started from a common ancestor process. Often this isn't very convenient, because you would like unrelated processes to be able to exchange data.

You do this with *FIFOs*, often referred to as *named pipes*. A named pipe is a special type of file (remember that everything in Linux is a file!) that exists as a name in the file system but behaves like the unnamed pipes that you've met already.

You can create named pipes from the command line and from within a program. Historically, the command-line program for creating them was `mknod`:

```
$ mknod filename p
```

However, the `mknod` command is not in the X/Open command list, so it may not be available on all UNIX-like systems. The preferred command-line method is to use

```
$ mkfifo filename
```

*Some older versions of UNIX only had the* `mknod` *command. X/Open Issue 4 Version 2 has the* `mknod` *function call, but not the command-line program. Linux, friendly as ever, supplies both* `mknod` *and* `mkfifo`*.*

From inside a program, you can use two different calls:

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

int mkfifo(const char *filename, mode_t mode);
int mknod(const char *filename, mode_t mode | S_IFIFO, (dev_t) 0);
```

Like the `mknod` command, you can use the `mknod` function for making many special types of files. Using a `dev_t` value of 0 and ORing the file access mode with `S_IFIFO` is the only portable use of this function that creates a named pipe. We'll use the simpler `mkfifo` function in the examples.

## Try It Out Creating a Named Pipe

The following example is `fifo1.c`:

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

int main()
{
    int res = mkfifo("/tmp/my_fifo", 0777);
    if (res == 0) printf("FIFO created\n");
    exit(EXIT_SUCCESS);
}
```

You can create and look for the pipe with

```
$ ./fifo1
FIFO created
$ ls -lF /tmp/my_fifo
prwxr-xr-x   1 rick     users           0 2007-06-16 17:18 /tmp/my_fifo|
```

Notice that the first character of output is a p, indicating a pipe. The | symbol at the end is added by the ls command's -F option and also indicates a pipe.

### How It Works

The program uses the mkfifo function to create a special file. Although you ask for a mode of 0777, this is altered by the user mask (umask) setting (in this case 022), just as in normal file creation, so the resulting file has mode 755. If your umask is set differently, for example to 0002, you will see different permissions on the created file.

You can remove the FIFO just like a conventional file by using the rm command, or from within a program by using the unlink system call.

## Accessing a FIFO

One very useful feature of named pipes is that, because they appear in the file system, you can use them in commands where you would normally use a filename. Before you do more programming using the FIFO file you created, let's investigate the behavior of the FIFO file using normal file commands.

### Try It Out Accessing a FIFO File

1. First, try reading the (empty) FIFO:

```
$ cat < /tmp/my_fifo
```

2. Now try writing to the FIFO. You will have to use a different terminal because the first command will now be hanging, waiting for some data to appear in the FIFO.

```
$ echo "Hello World" > /tmp/my_fifo
```

   You will see the output appear from the cat command. If you don't send any data down the FIFO, the cat command will hang until you interrupt it, conventionally with Ctrl+C.

3. You can do both at once by putting the first command in the background:

```
$ cat < /tmp/my_fifo &
[1] 1316
$ echo "Hello World" > /tmp/my_fifo
Hello World

[1]+  Done                    cat </tmp/my_fifo
$
```

### How It Works

Because there was no data in the FIFO, the cat and echo programs both block, waiting for some data to arrive and some other process to read the data, respectively.

Looking at the third stage, the cat process is initially blocked in the background. When echo makes some data available, the cat command reads the data and prints it to the standard output. Notice that the `cat` program then exits without waiting for more data. It doesn't block because the pipe will have been closed when the second command putting data in the FIFO completed, so calls to `read` in the `cat` program will return 0 bytes, indicating the end of file.

Now that you've seen how the FIFO behaves when you access it using command-line programs, let's look in more detail at the program interface, which allows you more control over how `read`s and `write`s behave when you're accessing a FIFO.

*Unlike a pipe created with the* `pipe` *call, a FIFO exists as a named file, not as an open file descriptor, and it must be opened before it can be read from or written to. You open and close a FIFO using the same* `open` *and* `close` *functions that you saw used earlier for files, with some additional functionality. The* `open` *call is passed the path name of the FIFO, rather than that of a regular file.*

## Opening a FIFO with open

The main restriction on opening FIFOs is that a program may not open a FIFO for reading and writing with the mode `O_RDWR`. If a program violates this restriction, the result is undefined. This is quite a sensible restriction because, normally, you use a FIFO only for passing data in a single direction, so there is no need for an `O_RDWR` mode. A process would read its own output back from a pipe if it were opened read/write.

If you do want to pass data in both directions between programs, it's much better to use either a pair of FIFOs or pipes, one for each direction, or (unusually) explicitly change the direction of the data flow by closing and reopening the FIFO. We return to bidirectional data exchange using FIFOs later in the chapter.

The other difference between opening a FIFO and a regular file is the use of the `open_flag` (the second parameter to `open`) with the option `O_NONBLOCK`. Using this `open` mode not only changes how the `open` call is processed, but also changes how `read` and `write` requests are processed on the returned file descriptor.

There are four legal combinations of `O_RDONLY`, `O_WRONLY`, and the `O_NONBLOCK` flag. We'll consider each in turn.

```
open(const char *path, O_RDONLY);
```

In this case, the `open` call will block; it will not return until a process opens the same FIFO for writing. This is like the first `cat` example.

```
open(const char *path, O_RDONLY | O_NONBLOCK);
```

The `open` call will now succeed and return immediately, even if the FIFO has not been opened for writing by any process.

```
open(const char *path, O_WRONLY);
```

In this case, the `open` call will block until a process opens the same FIFO for reading.

```
open(const char *path, O_WRONLY | O_NONBLOCK);
```

This will always return immediately, but if no process has the FIFO open for reading, `open` will return an error, –1, and the FIFO won't be opened. If a process does have the FIFO open for reading, the file descriptor returned can be used for writing to the FIFO.

*Notice the asymmetry between the use of* `O_NONBLOCK` *with* `O_RDONLY` *and* `O_WRONLY`*, in that a non-blocking* `open` *for writing fails if no process has the pipe open for reading, but a nonblocking* `read` *doesn't fail. The behavior of the* `close` *call isn't affected by the* `O_NONBLOCK` *flag.*

## Try It Out Opening FIFO Files

Now look at how you can use the behavior of `open` with the `O_NONBLOCK` flag to synchronize two processes. Rather than use a number of example programs, you'll write a single test program, `fifo2.c`, which allows you to investigate the behavior of FIFOs by passing in different parameters.

1. Start with the header files, a `#define`, and the check that the correct number of command-line arguments has been supplied:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define FIFO_NAME "/tmp/my_fifo"

int main(int argc, char *argv[])
{
    int res;
    int open_mode = 0;
    int i;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s <some combination of\
               O_RDONLY O_WRONLY O_NONBLOCK>\n", *argv);
        exit(EXIT_FAILURE);
    }
```

2. Assuming that the program passed the test, you now set the value of `open_mode` from those arguments:

```
    for(i = 1; i <argc; i++) {
        if (strncmp(*++argv, "O_RDONLY", 8) == 0)
            open_mode |= O_RDONLY;
        if (strncmp(*argv, "O_WRONLY", 8) == 0)
            open_mode |= O_WRONLY;
        if (strncmp(*argv, "O_NONBLOCK", 10) == 0)
            open_mode |= O_NONBLOCK;
    }
```

3. Next check whether the FIFO exists, and create it if necessary. Then the FIFO is opened and output given to that effect while the program catches forty winks. Last of all, the FIFO is closed.

```
    if (access(FIFO_NAME, F_OK) == -1) {
        res = mkfifo(FIFO_NAME, 0777);
        if (res != 0) {
            fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
            exit(EXIT_FAILURE);
        }
    }

    printf("Process %d opening FIFO\n", getpid());
    res = open(FIFO_NAME, open_mode);
    printf("Process %d result %d\n", getpid(), res);
    sleep(5);
    if (res != -1) (void)close(res);
    printf("Process %d finished\n", getpid());
    exit(EXIT_SUCCESS);
}
```

### How It Works

This program allows you to specify on the command line the combinations of `O_RDONLY`, `O_WRONLY`, and `O_NONBLOCK` that you want to use. It does this by comparing known strings with command-line parameters and setting (with `|=`) the appropriate flag if the string matches. The program uses the `access` function to check whether the FIFO file already exists and will create it if required.

You never destroy the FIFO, because you have no way of telling if another program already has the FIFO in use.

## O_RDONLY and O_WRONLY without O_NONBLOCK

You now have your test program, so you can try out a couple of combinations. Notice that the first program, the reader, has been put in the background:

```
$ ./fifo2 O_RDONLY &
[1] 152
Process 152 opening FIFO
$ ./fifo2 O_WRONLY
Process 153 opening FIFO
Process 152 result 3
Process 153 result 3
Process 152 finished
Process 153 finished
```

This is probably the most common use of named pipes. It allows the reader process to start and wait in the `open` call and then allows both programs to continue when the second program opens the FIFO. Notice that both the reader and writer processes have synchronized at the `open` call.

*When a Linux process is blocked, it doesn't consume CPU resources, so this method of process synchronization is very CPU-efficient.*

### *O_RDONLY with O_NONBLOCK and O_WRONLY*

In the following example, the reader process executes the `open` call and continues immediately, even though no writer process is present. The writer also immediately continues past the `open` call, because the FIFO is already open for reading.

```
$ ./fifo2 O_RDONLY O_NONBLOCK &
[1] 160
Process 160 opening FIFO
$ ./fifo2 O_WRONLY
Process 161 opening FIFO
Process 160 result 3
Process 161 result 3
Process 160 finished
Process 161 finished
[1]+  Done                    ./fifo2 O_RDONLY O_NONBLOCK
```

These two examples are probably the most common combinations of `open` modes. Feel free to use the example program to experiment with some other combinations.

## *Reading and Writing FIFOs*

Using the `O_NONBLOCK` mode affects how `read` and `write` calls behave on FIFOs.

A `read` on an empty blocking FIFO (that is, one not opened with `O_NONBLOCK`) will wait until some data can be read. Conversely, a `read` on a nonblocking FIFO with no data will return 0 bytes.

A `write` on a full blocking FIFO will wait until the data can be written. A `write` on a FIFO that can't accept all of the bytes being written will either:

❑ Fail, if the request is for `PIPE_BUF` bytes or less and the data can't be written.

❑ Write part of the data, if the request is for more than `PIPE_BUF` bytes, returning the number of bytes actually written, which could be 0.

The size of a FIFO is an important consideration. There is a system-imposed limit on how much data can be "in" a FIFO at any one time. This is the `#define PIPE_BUF`, usually found in `limits.h`. On Linux and many other UNIX-like systems, this is commonly 4,096 bytes, but it could be as low as 512 bytes on some systems. The system guarantees that writes of `PIPE_BUF` or fewer bytes on a FIFO that has been opened `O_WRONLY` (that is, blocking) will either write all or none of the bytes.

Although this limit is not very important in the simple case of a single FIFO writer and a single FIFO reader, it's quite common to use a single FIFO to allow many different programs to send requests to a single FIFO reader. If several different programs try to write to the FIFO at the same time, it's usually vital that the blocks of data from different programs don't get interleaved — that is, each `write` must be "atomic." How do you do this?

Well, if you ensure that all your `write` requests are to a blocking FIFO and are less than `PIPE_BUF` bytes in size, the system will ensure that data never gets interleaved. In general, it's a good idea to restrict the data transferred via a FIFO to blocks of `PIPE_BUF` bytes, unless you're using only a single-writer and a single-reader process.

## Try It Out Inter-Process Communication with FIFOs

To show how unrelated processes can communicate using named pipes, you need two separate programs, `fifo3.c` and `fifo4.c`.

1. The first program is the producer program. It creates the pipe if required, and then writes data to it as quickly as possible.

*Note that, for illustration purposes, we don't mind what the data is, so we don't bother to initialize a buffer. In both listings, shaded lines show the changes from* `fifo2.c`*, with all the command-line argument code removed.*

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>

#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
#define TEN_MEG (1024 * 1024 * 10)

int main()
{
    int pipe_fd;
    int res;
    int open_mode = O_WRONLY;
    int bytes_sent = 0;
    char buffer[BUFFER_SIZE + 1];

    if (access(FIFO_NAME, F_OK) == -1) {
        res = mkfifo(FIFO_NAME, 0777);
        if (res != 0) {
            fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
            exit(EXIT_FAILURE);
        }
    }

    printf("Process %d opening FIFO O_WRONLY\n", getpid());
    pipe_fd = open(FIFO_NAME, open_mode);
    printf("Process %d result %d\n", getpid(), pipe_fd);

    if (pipe_fd != -1) {
        while(bytes_sent < TEN_MEG) {
            res = write(pipe_fd, buffer, BUFFER_SIZE);
            if (res == -1) {
                fprintf(stderr, "Write error on pipe\n");
                exit(EXIT_FAILURE);
            }
```

```
                bytes_sent += res;
            }
            (void)close(pipe_fd);
        }
        else {
            exit(EXIT_FAILURE);
        }

        printf("Process %d finished\n", getpid());
        exit(EXIT_SUCCESS);
}
```

2. The second program, the consumer, is much simpler. It reads and discards data from the FIFO.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>


#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF

int main()
{
    int pipe_fd;
    int res;
    int open_mode = O_RDONLY;
    char buffer[BUFFER_SIZE + 1];
    int bytes_read = 0;

    memset(buffer, '\0', sizeof(buffer));

    printf("Process %d opening FIFO O_RDONLY\n", getpid());
    pipe_fd = open(FIFO_NAME, open_mode);
    printf("Process %d result %d\n", getpid(), pipe_fd);

    if (pipe_fd != -1) {
        do {
            res = read(pipe_fd, buffer, BUFFER_SIZE);
            bytes_read += res;
        } while (res > 0);
        (void)close(pipe_fd);
    }
    else {
        exit(EXIT_FAILURE);
    }

    printf("Process %d finished, %d bytes read\n", getpid(), bytes_read);
    exit(EXIT_SUCCESS);
}
```

When you run these programs at the same time, using the `time` command to time the reader, the output you get (with some tidying for clarity) is

```
$ ./fifo3 &
[1] 375
Process 375 opening FIFO O_WRONLY
$ time ./fifo4
Process 377 opening FIFO O_RDONLY
Process 375 result 3
Process 377 result 3
Process 375 finished
Process 377 finished, 10485760 bytes read

real    0m0.053s
user    0m0.020s
sys     0m0.040s

[1]+  Done                    ./fifo3
```

### How It Works

Both programs use the FIFO in blocking mode. You start `fifo3` (the writer/producer) first, which blocks, waiting for a reader to open the FIFO. When `fifo4` (the consumer) is started, the writer is then unblocked and starts writing data to the pipe. At the same time, the reader starts reading data from the pipe.

*Linux arranges the scheduling of the two processes so that they both run when they can and are blocked when they can't. Thus, the writer is blocked when the pipe is full, and the reader is blocked when the pipe is empty.*

The output from the `time` command shows that it took the reader well under one-tenth of a second to run, reading 10 megabytes of data in the process. This shows that pipes, at least as implemented in modern versions of Linux, can be an efficient way of transferring data between programs.

## Advanced Topic: Client/Server Using FIFOs

For your final look at FIFOs, let's consider how you might build a very simple client/server application using named pipes. You want to have a single-server process that accepts requests, processes them, and returns the resulting data to the requesting party: the client.

You want to allow multiple client processes to send data to the server. In the interests of simplicity, we'll assume that the data to be processed can be broken into blocks, each smaller than `PIPE_BUF` bytes. Of course, you could implement this system in many ways, but we'll consider only one method as an illustration of how named pipes can be used.

Because the server will process only one block of information at a time, it seems logical to have a single FIFO that is read by the server and written to by each of the clients. By opening the FIFO in blocking mode, the server and the clients will be automatically blocked as required.

Returning the processed data to the clients is slightly more difficult. You need to arrange a second pipe, one per client, for the returned data. By passing the process identifier (PID) of the client in the original data sent to the server, both parties can use this to generate the unique name for the return pipe.

## Try It Out An Example Client/Server Application

1. First, you need a header file, `client.h`, that defines the data common to both client and server programs. It also includes the required system headers, for convenience.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>

#define SERVER_FIFO_NAME "/tmp/serv_fifo"
#define CLIENT_FIFO_NAME "/tmp/cli_%d_fifo"

#define BUFFER_SIZE 20

struct data_to_pass_st {
    pid_t  client_pid;
    char   some_data[BUFFER_SIZE - 1];
};
```

2. Now for the server program, `server.c`. In this section, you create and then open the server pipe. It's set to be read-only, with blocking. After sleeping (for demonstration purposes), the server reads in any data from the client, which has the `data_to_pass_st` structure.

```
#include "client.h"
#include <ctype.h>

int main()
{
    int server_fifo_fd, client_fifo_fd;
    struct data_to_pass_st my_data;
    int read_res;
    char client_fifo[256];
    char *tmp_char_ptr;

    mkfifo(SERVER_FIFO_NAME, 0777);
    server_fifo_fd = open(SERVER_FIFO_NAME, O_RDONLY);
    if (server_fifo_fd == -1) {
        fprintf(stderr, "Server fifo failure\n");
        exit(EXIT_FAILURE);
    }

    sleep(10); /* lets clients queue for demo purposes */

    do {
        read_res = read(server_fifo_fd, &my_data, sizeof(my_data));
        if (read_res > 0) {
```

3. In this next stage, you perform some processing on the data just read from the client: Convert all the characters in `some_data` to uppercase and combine the `CLIENT_FIFO_NAME` with the received `client_pid`.

```
            tmp_char_ptr = my_data.some_data;
            while (*tmp_char_ptr) {
                *tmp_char_ptr = toupper(*tmp_char_ptr);
                tmp_char_ptr++;
            }
            sprintf(client_fifo, CLIENT_FIFO_NAME, my_data.client_pid);
```

4. Then send the processed data back, opening the client pipe in write-only, blocking mode. Finally, shut down the server FIFO by closing the file and then `unlink`ing the FIFO.

```
            client_fifo_fd = open(client_fifo, O_WRONLY);
            if (client_fifo_fd != -1) {
                write(client_fifo_fd, &my_data, sizeof(my_data));
                close(client_fifo_fd);
            }
        }
    } while (read_res > 0);
    close(server_fifo_fd);
    unlink(SERVER_FIFO_NAME);
    exit(EXIT_SUCCESS);
}
```

5. Here's the client, `client.c`. The first part of this program opens the server FIFO, if it already exists, as a file. It then gets its own process ID, which forms some of the data that will be sent to the server. The client FIFO is created, ready for the next section.

```
#include "client.h"
#include <ctype.h>

int main()
{
    int server_fifo_fd, client_fifo_fd;
    struct data_to_pass_st my_data;
    int times_to_send;
    char client_fifo[256];

    server_fifo_fd = open(SERVER_FIFO_NAME, O_WRONLY);
    if (server_fifo_fd == -1) {
        fprintf(stderr, "Sorry, no server\n");
        exit(EXIT_FAILURE);
    }

    my_data.client_pid = getpid();
    sprintf(client_fifo, CLIENT_FIFO_NAME, my_data.client_pid);
    if (mkfifo(client_fifo, 0777) == -1) {
        fprintf(stderr, "Sorry, can't make %s\n", client_fifo);
        exit(EXIT_FAILURE);
    }
```

6. For each of the five loops, the client data is sent to the server. Then the client FIFO is opened (read-only, blocking mode) and the data read back. Finally, the server FIFO is closed and the client FIFO removed from the file system.

```
        for (times_to_send = 0; times_to_send < 5; times_to_send++) {
            sprintf(my_data.some_data, "Hello from %d", my_data.client_pid);
            printf("%d sent %s, ", my_data.client_pid, my_data.some_data);
            write(server_fifo_fd, &my_data, sizeof(my_data));
            client_fifo_fd = open(client_fifo, O_RDONLY);
            if (client_fifo_fd != -1) {
                if (read(client_fifo_fd, &my_data, sizeof(my_data)) > 0) {
                    printf("received: %s\n", my_data.some_data);
                }
                close(client_fifo_fd);
            }
        }
        close(server_fifo_fd);
        unlink(client_fifo);
        exit(EXIT_SUCCESS);
    }
```

To test this application, you need to run a single copy of the server and several clients. To get them all started at close to the same time, use the following shell commands:

```
$ ./server &
$ for i in 1 2 3 4 5
do
./client &
done
$
```

This starts one server process and five client processes. The output from the clients, edited for brevity, looks like this:

```
531 sent Hello from 531, received: HELLO FROM 531
532 sent Hello from 532, received: HELLO FROM 532
529 sent Hello from 529, received: HELLO FROM 529
530 sent Hello from 530, received: HELLO FROM 530
531 sent Hello from 531, received: HELLO FROM 531
532 sent Hello from 532, received: HELLO FROM 532
```

As you can see in this output, different client requests are being interleaved, but each client is getting the suitably processed data returned to it. Note that you may or may not see this interleaving; the order in which client requests are received may vary between machines and possibly between runs on the same machine.

## *How It Works*

Now we'll cover the sequence of client and server operations as they interact, something that we haven't covered so far.

The server creates its FIFO in read-only mode and blocks. It does this until the first client connects by opening the same FIFO for writing. At that point, the server process is unblocked and the `sleep` is executed, so the `write`s from the clients queue up. (In a real application, the `sleep` would be removed; we're only using it to demonstrate the correct operation of the program with multiple simultaneous clients.)

In the meantime, after the client has opened the server FIFO, it creates its own uniquely named FIFO for reading data back from the server. Only then does the client write data to the server (blocking if the pipe is full or the server's still sleeping) and then blocks on a `read` of its own FIFO, waiting for the reply.

On receiving the data from the client, the server processes it, opens the client pipe for writing, and writes the data back, which unblocks the client. When the client is unblocked, it can read from its pipe the data written to it by the server.

The whole process repeats until the last client closes the server pipe, causing the server's `read` to fail (returning 0) because no process has the server pipe open for writing. If this were a real server process that needed to wait for further clients, you would need to modify it to either

- ❑ Open a file descriptor to its own server pipe, so `read` always blocks rather than returning 0.
- ❑ Close and reopen the server pipe when `read` returns 0 bytes, so the server process blocks in the `open` waiting for a client, just as it did when it first started.

Both of these techniques are illustrated in the rewrite of the CD database application to use named pipes.

## The CD Database Application

Now that you've seen how you can use named pipes to implement a simple client/server system, you can revisit the CD database application and convert it accordingly. You'll also incorporate some signal handling to allow you to perform some tidy-up actions when the process is interrupted. You will use the earlier `dbm` version of the application that had a command-line interface to see the code as straightforwardly as possible.

Before you get to look in detail at this new version, you must compile the application. If you have the source code from the website, use the `makefile` to compile it into the `server` and `client` programs.

*As you saw early in Chapter 7, different distributions name and install the* `dbm` *files in slightly different ways. If the provided files do not compile on your distribution, check back to Chapter 7 for further advice on the naming and location of the* `dbm` *files.*

Running `server -i` allows the program to initialize a new CD database.

Needless to say, the client won't run unless the server is up and running. Here's the `makefile` to show how the programs fit together:

```
all:     server client
```

```
CC=cc
CFLAGS= -pedantic -Wall

# For debugging un-comment the next line
# DFLAGS=-DDEBUG_TRACE=1 -g

# Where, and which version, of dbm are we using.
# This assumes gdbm is pre-installed in a standard place, but we are
# going to use the gdbm compatibility routines, that make it emulate ndbm.
# We do this because ndbm is the 'most standard' of the dbm versions.
# Depending on your distribution, these may need changing.
DBM_INC_PATH=/usr/include/gdbm
DBM_LIB_PATH=/usr/lib
DBM_LIB_FILE=-lgdbm
# On some distributions you may need to change the above line to include
# the compatibility library, as shown below.
# DBM_LIB_FILE=-lgdbm_compat -lgdbm

.c.o:
	$(CC) $(CFLAGS) -I$(DBM_INC_PATH) $(DFLAGS) -c $<

app_ui.o: app_ui.c cd_data.h
cd_dbm.o: cd_dbm.c cd_data.h
client_f.o: clientif.c cd_data.h cliserv.h
pipe_imp.o: pipe_imp.c cd_data.h cliserv.h
server.o: server.c cd_data.h cliserv.h

client: app_ui.o clientif.o pipe_imp.o
	$(CC) -o client  $(DFLAGS) app_ui.o clientif.o pipe_imp.o

server:  server.o cd_dbm.o pipe_imp.o
	$(CC) -o server -L$(DBM_LIB_PATH) $(DFLAGS) server.o cd_dbm.o pipe_imp.o -
l$(DBM_LIB_FILE)

clean:
	rm -f server client_app *.o *~
```

## Aims

The aim is to split the part of the application that deals with the database away from the user interface part of the application. You also want to run a single-server process, but allow many simultaneous clients, and to minimize changes to the existing code. Wherever possible, you will leave existing code unchanged.

To keep things simple, you also want to be able to create (and delete) pipes within the application, so there's no need for a system administrator to create named pipes before you can use them.

It's also important to ensure that you never "busy wait," wasting CPU time, for an event. As you've seen, Linux allows you to block, waiting for events without using significant resources. You should use the blocking nature of pipes to ensure that you use the CPU efficiently. After all, the server could, in theory, wait for many hours for a request to arrive.

## Implementation

The earlier, single-process version of the application that you saw in Chapter 7 used a set of data access routines for manipulating the data. These were

```
int database_initialize(const int new_database);
void database_close(void);
cdc_entry get_cdc_entry(const char *cd_catalog_ptr);
cdt_entry get_cdt_entry(const char *cd_catalog_ptr, const int track_no);
int add_cdc_entry(const cdc_entry entry_to_add);
int add_cdt_entry(const cdt_entry entry_to_add);
int del_cdc_entry(const char *cd_catalog_ptr);
int del_cdt_entry(const char *cd_catalog_ptr, const int track_no);
cdc_entry search_cdc_entry(const char *cd_catalog_ptr,
                           int *first_call_ptr);
```

These functions provide a convenient place to make a clean separation between client and server.

In the single-process implementation, you can view the application as having two parts, even though it was compiled as a single program, as shown in Figure 13-6.

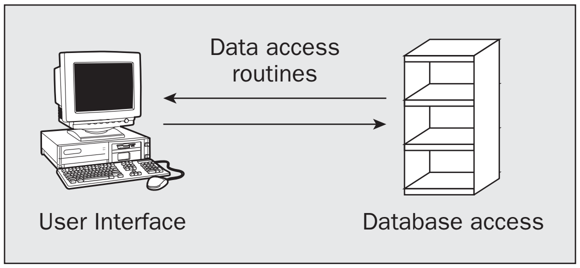

**Figure 13-6**

In the client-server implementation, you want to insert some named pipes and supporting code between the two major parts of the application. Figure 13-7 shows the structure you need.

In the implementation, both the client and server interface routines are put in the same file, `pipe_imp.c`. This keeps all the code that depends on the use of named pipes for the client/server implementation in a single file. The formatting and packaging of the data being passed is kept separate from the routines that implement the named pipes. You end up with more source files, but a better logical division between them. The calling structure in the application is illustrated in Figure 13-8.

The files `app_ui.c`, `client_if.c`, and `pipe_imp.c` are compiled and linked together to give a client program. The files `cd_dbm.c`, `server.c`, and `pipe_imp.c` are compiled and linked together to give a server program. A header file, `cliserv.h`, acts as a common definitions header file to tie the two together.

The files `app_ui.c` and `cd_dbm.c` have only very minor changes, principally to allow for the split into two programs. Because the application is now quite large and a significant proportion of the code is unchanged from that previously seen, we show here only the files `cliserv.h`, `client_if.c`, and `pipe_imp.c`.

*Some parts of this file are dependent on the specific client/server implementation, in this case named pipes. We'll be changing to a different client/server model at the end of Chapter 14.*

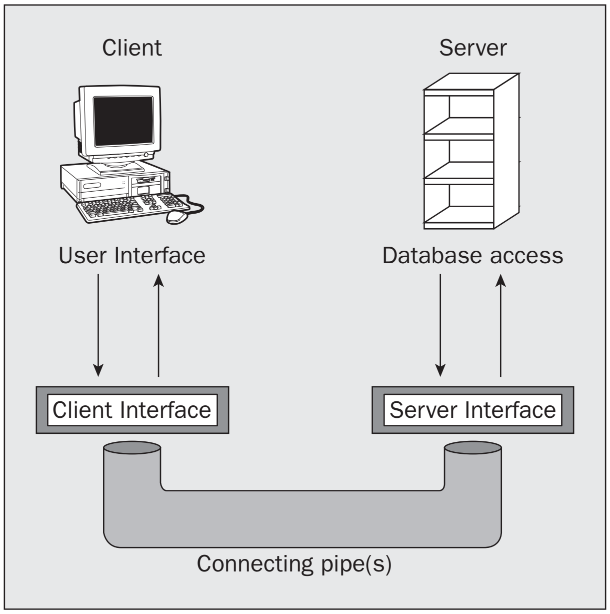

Figure 13-7

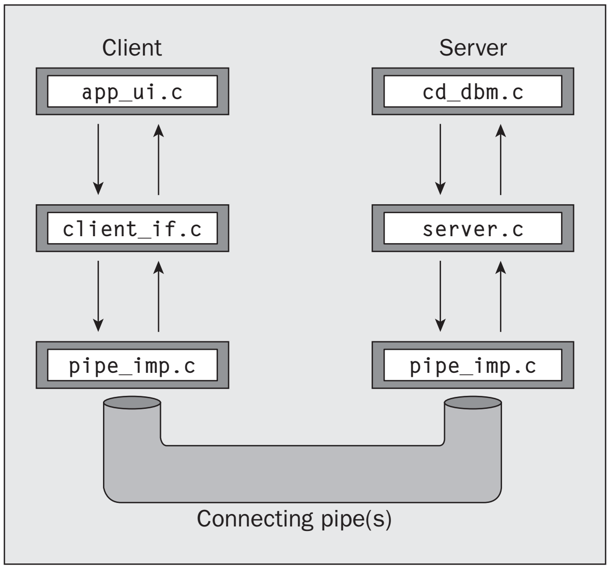

Figure 13-8

### The Header File, cliserv.h

First look at `cliserv.h`. This file defines the client/server interface. It's required by both client and server implementations.

1. Following are the required `#include` headers:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
```

2. You then define the named pipes. Use one pipe for the server and one pipe for each client. Because there may be multiple clients, the client incorporates a process ID into the name to ensure that its pipe is unique:

```
#define SERVER_PIPE "/tmp/server_pipe"
#define CLIENT_PIPE "/tmp/client_%d_pipe"

#define ERR_TEXT_LEN 80
```

3. Implement the commands as enumerated types, rather than `#define`s.

*This is a good way of allowing the compiler to do more type checking and also helps in debugging the application, because many debuggers are able to show the name of enumerated constants, but not the name defined by a `#define` directive.*

The first `typedef` gives the type of request being sent to the server; the second gives the server response to the client:

```
typedef enum {
    s_create_new_database = 0,
    s_get_cdc_entry,
    s_get_cdt_entry,
    s_add_cdc_entry,
    s_add_cdt_entry,
    s_del_cdc_entry,
    s_del_cdt_entry,
    s_find_cdc_entry
} client_request_e;

typedef enum {
    r_success = 0,
    r_failure,
    r_find_no_more
} server_response_e;
```

4. Next, declare a structure that will form the message passed in both directions between the two processes.

*Because you don't actually need to return both a* `cdc_entry` *and* `cdt_entry` *in the same response, you could have combined them in a union. However, for simplicity you can keep them separate. This also makes the code easier to maintain.*

```
typedef struct {
    pid_t               client_pid;
    client_request_e    request;
    server_response_e   response;
    cdc_entry           cdc_entry_data;
    cdt_entry           cdt_entry_data;
    char                error_text[ERR_TEXT_LEN + 1];
} message_db_t;
```

5. Finally, here are the pipe interface functions that perform data transfer, implemented in `pipe_imp.c`. These divide into server- and client-side functions, in the first and second blocks, respectively:

```
int server_starting(void);
void server_ending(void);
int read_request_from_client(message_db_t *rec_ptr);
int start_resp_to_client(const message_db_t mess_to_send);
int send_resp_to_client(const message_db_t mess_to_send);
void end_resp_to_client(void);

int client_starting(void);
void client_ending(void);
int send_mess_to_server(message_db_t mess_to_send);
int start_resp_from_server(void);
int read_resp_from_server(message_db_t *rec_ptr);
void end_resp_from_server(void);
```

We split the rest of the discussion into the client interface functions and details of the server- and client-side functions found in `pipe_imp.c`, and we look at the source code as necessary.

## Client Interface Functions

Now look at `clientif.c`. This provides "fake" versions of the database access routines. These encode the request in a `message_db_t` structure and then use the routines in `pipe_imp.c` to transfer the request to the server. This allows you to make minimal changes to the original `app_ui.c`.

### The Client's Interpreter

1. This file implements the nine database functions prototyped in `cd_data.h`. It does so by passing requests to the server and then returning the server response from the function, acting as an intermediary. The file starts with `#include` files and constants:

```
#define _POSIX_SOURCE
```

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>

#include "cd_data.h"
#include "cliserv.h"
```

2. The static variable `mypid` reduces the number of calls to `getpid` that would otherwise be required. We use a local function, `read_one_response`, to eliminate duplicated code:

```
static pid_t mypid;

static int read_one_response(message_db_t *rec_ptr);
```

3. The `database_initialize` and `close` routines are still called, but are now used, respectively, for initializing the client side of the pipes interface and for removing redundant named pipes when the client exits:

```
int database_initialize(const int new_database)
{
    if (!client_starting()) return(0);
    mypid = getpid();
    return(1);

} /* database_initialize */

void database_close(void) {
    client_ending();
}
```

4. The `get_cdc_entry` routine is called to get a catalog entry from the database, given a CD catalog title. Here you encode the request in a `message_db_t` structure and pass it to the server. You then read the response back into a different `message_db_t` structure. If an entry is found, it's included inside the `message_db_t` structure as a `cdc_entry` structure, so you pass back the appropriate part of the structure:

```
cdc_entry get_cdc_entry(const char *cd_catalog_ptr)
{
    cdc_entry ret_val;
    message_db_t mess_send;
    message_db_t mess_ret;

    ret_val.catalog[0] = '\0';
    mess_send.client_pid = mypid;
    mess_send.request = s_get_cdc_entry;
    strcpy(mess_send.cdc_entry_data.catalog, cd_catalog_ptr);

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                ret_val = mess_ret.cdc_entry_data;
            } else {
                fprintf(stderr, "%s", mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(ret_val);
}
```

5. Here's the source for the function `read_one_response` that you use to avoid duplicating code:

```
static int read_one_response(message_db_t *rec_ptr) {

    int return_code = 0;
    if (!rec_ptr) return(0);

    if (start_resp_from_server()) {
        if (read_resp_from_server(rec_ptr)) {
            return_code = 1;
        }
        end_resp_from_server();
    }
    return(return_code);
}
```

6. The other `get_xxx`, `del_xxx`, and `add_xxx` routines are implemented in a similar way to the `get_cdc_entry` function and are reproduced here for completeness. First, the function for retrieving CD tracks:

```
cdt_entry get_cdt_entry(const char *cd_catalog_ptr, const int track_no)
{
    cdt_entry ret_val;
    message_db_t mess_send;
    message_db_t mess_ret;

    ret_val.catalog[0] = '\0';
    mess_send.client_pid = mypid;
    mess_send.request = s_get_cdt_entry;
    strcpy(mess_send.cdt_entry_data.catalog, cd_catalog_ptr);
    mess_send.cdt_entry_data.track_no = track_no;

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                ret_val = mess_ret.cdt_entry_data;
            } else {
                fprintf(stderr, "%s", mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(ret_val);
}
```

7. Next, two functions for adding data, first to the catalog and then to the tracks database:

```
int add_cdc_entry(const cdc_entry entry_to_add)
{
    message_db_t mess_send;
    message_db_t mess_ret;

    mess_send.client_pid = mypid;
    mess_send.request = s_add_cdc_entry;
    mess_send.cdc_entry_data = entry_to_add;

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                return(1);
            } else {
                fprintf(stderr, "%s", mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(0);
}

int add_cdt_entry(const cdt_entry entry_to_add)
{
    message_db_t mess_send;
    message_db_t mess_ret;

    mess_send.client_pid = mypid;
    mess_send.request = s_add_cdt_entry;
    mess_send.cdt_entry_data = entry_to_add;

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                return(1);
            } else {
                fprintf(stderr, "%s", mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(0);
}
```

**8.** Last, two functions for data deletion:

```
int del_cdc_entry(const char *cd_catalog_ptr)
{
    message_db_t mess_send;
    message_db_t mess_ret;

    mess_send.client_pid = mypid;
    mess_send.request = s_del_cdc_entry;
    strcpy(mess_send.cdc_entry_data.catalog, cd_catalog_ptr);

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                return(1);
            } else {
                fprintf(stderr, "%s", mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(0);
}

int del_cdt_entry(const char *cd_catalog_ptr, const int track_no)
{
    message_db_t mess_send;
    message_db_t mess_ret;

    mess_send.client_pid = mypid;
    mess_send.request = s_del_cdt_entry;
    strcpy(mess_send.cdt_entry_data.catalog, cd_catalog_ptr);
    mess_send.cdt_entry_data.track_no = track_no;

    if (send_mess_to_server(mess_send)) {
        if (read_one_response(&mess_ret)) {
            if (mess_ret.response == r_success) {
                return(1);
            } else {
                fprintf(stderr, "%s", mess_ret.error_text);
            }
        } else {
            fprintf(stderr, "Server failed to respond\n");
        }
    } else {
        fprintf(stderr, "Server not accepting requests\n");
    }
    return(0);
}
```

## Searching the Database

The function for the search on the CD key is more complex. The user of this function expects to call it once to start a search. We catered to this expectation in Chapter 7 by setting `*first_call_ptr` to `true` on this first call and the function then to return the first match. On subsequent calls to the search function, `*first_call_ptr` is `false` and further matches are returned, one per call.

Now that you've split the application across two processes, you can no longer allow the search to proceed one entry at a time in the server, because a different client may request a different search from the server while your search is in progress. You can't make the server side store the context (how far the search has gotten) for each client search separately, because the client side can simply stop searching part of the way through a search, when a user finds the CD he is looking for or if the client "falls over."

You can either change the way the search is performed or, as in the following code, hide the complexity in the interface routine. This code arranges for the server to return all the possible matches to a search and then store them in a temporary file until the client requests them.

1. This function looks more complicated than it is because it calls three pipe functions that you'll be looking at in the next section: `send_mess_to_server`, `start_resp_from_server`, and `read_resp_from_server`.

```
    cdc_entry search_cdc_entry(const char *cd_catalog_ptr, int *first_call_ptr)
    {
        message_db_t mess_send;
        message_db_t mess_ret;

        static FILE *work_file = (FILE *)0;
        static int entries_matching = 0;
        cdc_entry ret_val;

        ret_val.catalog[0] = '\0';

        if (!work_file && (*first_call_ptr == 0)) return(ret_val);
```

2. Here's the first call to search, that is, with `*first_call_ptr` set to `true`. It's set to `false` immediately, in case you forget. A `work_file` is created and the client message structure initialized.

```
        if (*first_call_ptr) {
            *first_call_ptr = 0;
            if (work_file) fclose(work_file);
            work_file = tmpfile();
            if (!work_file) return(ret_val);

            mess_send.client_pid = mypid;
            mess_send.request = s_find_cdc_entry;
            strcpy(mess_send.cdc_entry_data.catalog, cd_catalog_ptr);
```

3. Next, there's this three-deep condition test, which makes calls to functions in `pipe_imp.c`. If the message is successfully sent to the server, the client waits for the server's response. While `read`s from the server are successful, the search matches are returned to the client's `work_file` and the `entries_matching` counter is incremented.

```
        if (send_mess_to_server(mess_send)) {
            if (start_resp_from_server()) {
                while (read_resp_from_server(&mess_ret)) {
                    if (mess_ret.response == r_success) {
                  fwrite(&mess_ret.cdc_entry_data, sizeof(cdc_entry), 1, work_file);
                        entries_matching++;
                    } else {
                        break;
                    }
                } /* while */
            } else {
                fprintf(stderr, "Server not responding\n");
            }
        } else {
            fprintf(stderr, "Server not accepting requests\n");
        }
```

4. The next test checks whether the search had any luck. Then the `fseek` call sets the `work_file` to the next place for data to be written.

```
        if (entries_matching == 0) {
            fclose(work_file);
            work_file = (FILE *)0;
            return(ret_val);
        }
        (void)fseek(work_file, 0L, SEEK_SET);
```

5. If this is not the first call to the search function with this particular search term, the code checks whether there are any matches left. Finally, the next matching entry is read to the `ret_val` structure. The previous checks guarantee that a matching entry exists.

```
    } else {
            /* not *first_call_ptr */
        if (entries_matching == 0) {
            fclose(work_file);
            work_file = (FILE *)0;
            return(ret_val);
        }
    }

    fread(&ret_val, sizeof(cdc_entry), 1, work_file);
    entries_matching--;

    return(ret_val);
}
```

## The Server Interface, server.c

Just as the client side has an interface to the `app_ui.c` program, so the server side needs a program to control the (renamed) `cd_access.c`, now `cd_dbm.c`. The server's `main` function is listed here.

1. Start by declaring some global variables, a prototype for the `process_command` function, and a signal-catcher function to ensure a clean exit:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <limits.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>

#include "cd_data.h"
#include "cliserv.h"

int save_errno;
static int server_running = 1;

static void process_command(const message_db_t mess_command);

void catch_signals()
{
    server_running = 0;
}
```

2. Now you come to the `main` function. After checking that the signal-catching routines are all right, the program checks to see whether you passed `-i` on the command line. If you did, it will create a new database. If the call to the `database_initialize` routine in `cd_dbm.c` fails, an error message is shown. If all is well and the server is running, any requests from the client are fed to the `process_command` function, which you'll see in a moment.

```
int main(int argc, char *argv[]) {
    struct sigaction new_action, old_action;
    message_db_t mess_command;
    int database_init_type = 0;

    new_action.sa_handler = catch_signals;
    sigemptyset(&new_action.sa_mask);
    new_action.sa_flags = 0;
    if ((sigaction(SIGINT, &new_action, &old_action) != 0) ||
        (sigaction(SIGHUP, &new_action, &old_action) != 0) ||
        (sigaction(SIGTERM, &new_action, &old_action) != 0)) {
        fprintf(stderr, "Server startup error, signal catching failed\n");
        exit(EXIT_FAILURE);
    }
    if (argc > 1) {
        argv++;
        if (strncmp("-i", *argv, 2) == 0) database_init_type = 1;
    }
    if (!database_initialize(database_init_type)) {
                fprintf(stderr, "Server error:-\
                         could not initialize database\n");
                exit(EXIT_FAILURE);
    }

    if (!server_starting()) exit(EXIT_FAILURE);

    while(server_running) {
        if (read_request_from_client(&mess_command)) {
            process_command(mess_command);
        } else {
            if(server_running) fprintf(stderr, "Server ended - can not \
                                        read pipe\n");
            server_running = 0;
        }
    } /* while */
    server_ending();
    exit(EXIT_SUCCESS);
}
```

3. Any client messages are fed to the `process_command` function, where they are fed into a `case` statement that makes the appropriate calls to `cd_dbm.c`:

```
static void process_command(const message_db_t comm)
{
    message_db_t resp;
    int first_time = 1;

    resp = comm; /* copy command back, then change resp as required */

    if (!start_resp_to_client(resp)) {
        fprintf(stderr, "Server Warning:-\
                 start_resp_to_client %d failed\n", resp.client_pid);
        return;
    }

    resp.response = r_success;
    memset(resp.error_text, '\0', sizeof(resp.error_text));
    save_errno = 0;

    switch(resp.request) {
        case s_create_new_database:
            if (!database_initialize(1)) resp.response = r_failure;
            break;
        case s_get_cdc_entry:
            resp.cdc_entry_data =
                           get_cdc_entry(comm.cdc_entry_data.catalog);
            break;
        case s_get_cdt_entry:
            resp.cdt_entry_data =
                           get_cdt_entry(comm.cdt_entry_data.catalog,
                                         comm.cdt_entry_data.track_no);
            break;
        case s_add_cdc_entry:
            if (!add_cdc_entry(comm.cdc_entry_data)) resp.response =
                           r_failure;
            break;
        case s_add_cdt_entry:
            if (!add_cdt_entry(comm.cdt_entry_data)) resp.response =
                           r_failure;
            break;
        case s_del_cdc_entry:
            if (!del_cdc_entry(comm.cdc_entry_data.catalog)) resp.response
                         = r_failure;
            break;
        case s_del_cdt_entry:
            if (!del_cdt_entry(comm.cdt_entry_data.catalog,
                 comm.cdt_entry_data.track_no)) resp.response = r_failure;
            break;
        case s_find_cdc_entry:
            do {
                resp.cdc_entry_data =
                          search_cdc_entry(comm.cdc_entry_data.catalog,
                                            &first_time);
                if (resp.cdc_entry_data.catalog[0] != 0) {
                    resp.response = r_success;
                    if (!send_resp_to_client(resp)) {
                        fprintf(stderr, "Server Warning:-\
                            failed to respond to %d\n", resp.client_pid);
                        break;
                    }
                } else {
                    resp.response = r_find_no_more;
                }
            } while (resp.response == r_success);
        break;
        default:
            resp.response = r_failure;
            break;
    } /* switch */

    sprintf(resp.error_text, "Command failed:\n\t%s\n",
             strerror(save_errno));

    if (!send_resp_to_client(resp)) {
        fprintf(stderr, "Server Warning:-\
                 failed to respond to %d\n", resp.client_pid);
    }

    end_resp_to_client();
    return;
}
```

Before you look at the actual pipe implementation, let's discuss the sequence of events that needs to occur to pass data between the client and server processes. Figure 13-9 shows both client and server processes starting and how both parties loop while processing commands and responses.

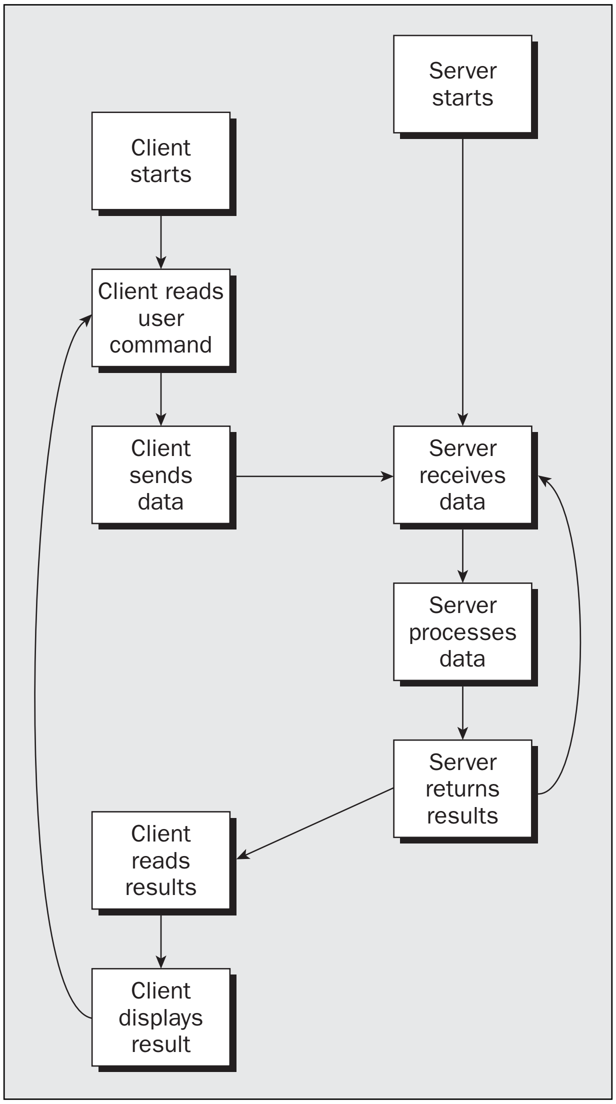

Figure 13-9

In this implementation, the situation is slightly more difficult, because, for a search request, the client passes a single command to the server and then expects to receive one or more responses from the server. This leads to some additional complexity, mainly in the client.

## The Pipe

Here's the pipes implementation file, pipe_imp.c, which has both the client- and server-side functions.

*As you saw in Chapter 10, the symbol* DEBUG_TRACE *can be defined to show the sequence of calls as the client and server processes pass messages to each other.*

### Pipes Implementation Header

1. First the #includes:

```
#include "cd_data.h"
#include "cliserv.h"
```

2. You also define some values that you need in different functions within the file:

```
static int server_fd = -1;
static pid_t mypid = 0;
static char client_pipe_name[PATH_MAX + 1] = {'\0'};
static int client_fd = -1;
static int client_write_fd = -1;
```

### Server-Side Functions

Next, you need to look at the server-side functions. The next section shows the functions that open and close the named pipe and read messages from the clients. The following section shows the code that opens, sends, and closes the client pipes based on the process ID the client includes in its message.

#### Server Functions

1. The server_starting routine creates the named pipe from which the server will read commands. It then opens that pipe for reading. This open will block until a client opens the pipe for writing. Use a blocking mode so that the server can perform blocking reads on the pipe while waiting for commands to be sent to it.

```
int server_starting(void)
{
    #if DEBUG_TRACE
        printf("%d :- server_starting()\n",  getpid());
    #endif

         unlink(SERVER_PIPE);
    if (mkfifo(SERVER_PIPE, 0777) == -1) {
        fprintf(stderr, "Server startup error, no FIFO created\n");
        return(0);
    }

    if ((server_fd = open(SERVER_PIPE, O_RDONLY)) == -1) {
        if (errno == EINTR) return(0);
        fprintf(stderr, "Server startup error, no FIFO opened\n");
        return(0);
    }
    return(1);
}
```

2. When the server ends, it removes the named pipe so that clients can detect that no server is running:

```
void server_ending(void)
{
    #if DEBUG_TRACE
        printf("%d :- server_ending()\n",  getpid());
    #endif

    (void)close(server_fd);
    (void)unlink(SERVER_PIPE);
}
```

3. The `read_request_from_client` function shown in the following example will block reading in the server pipe until a client writes a message into it:

```
int read_request_from_client(message_db_t *rec_ptr)
{
    int return_code = 0;
    int read_bytes;

    #if DEBUG_TRACE
        printf("%d :- read_request_from_client()\n",  getpid());
    #endif

    if (server_fd != -1) {
        read_bytes = read(server_fd, rec_ptr, sizeof(*rec_ptr));
```

```
...
```

```
    }
    return(return_code);
}
```

4. In the special case where no clients have the pipe open for writing, the `read` will return 0; that is, it detects an `EOF`. Then the server closes the pipe and opens it again, so that it blocks until a client also opens the pipe. This is just the same as when the server first starts; you have reinitialized the server. Insert this code into the preceding function:

```
        if (read_bytes == 0) {
            (void)close(server_fd);
            if ((server_fd = open(SERVER_PIPE, O_RDONLY)) == -1) {
                if (errno != EINTR) {
                    fprintf(stderr, "Server error, FIFO open failed\n");
                }
                return(0);
            }
            read_bytes = read(server_fd, rec_ptr, sizeof(*rec_ptr));
        }
        if (read_bytes == sizeof(*rec_ptr)) return_code = 1;
```

The server is a single process that may be serving many clients simultaneously. Because each client uses a different pipe to receive its responses, the server needs to write to a different pipe to send responses to different clients. Because file descriptors are a limited resource, the server opens a client pipe for writing only when it has data to send.

The code splits the opening, writing, and closing of the client pipe into three separate functions. You need to do this when you're returning multiple results to a search, so you can open the pipe once, write many responses, and close it again.

### Plumbing the Pipes

1. First open the client pipe:

```
int start_resp_to_client(const message_db_t mess_to_send)
{
    #if DEBUG_TRACE
        printf("%d :- start_resp_to_client()\n", getpid());
    #endif

    (void)sprintf(client_pipe_name, CLIENT_PIPE, mess_to_send.client_pid);
    if ((client_fd = open(client_pipe_name, O_WRONLY)) == -1) return(0);
    return(1);
}
```

2. The messages are all sent using calls to this function. You'll see the corresponding client-side functions that field the message later.

```
int send_resp_to_client(const message_db_t mess_to_send)
{
    int write_bytes;

    #if DEBUG_TRACE
        printf("%d :- send_resp_to_client()\n", getpid());
    #endif

    if (client_fd == -1) return(0);
    write_bytes = write(client_fd, &mess_to_send, sizeof(mess_to_send));
    if (write_bytes != sizeof(mess_to_send)) return(0);
    return(1);
}
```

3. Finally, close the client pipe:

```
void end_resp_to_client(void)
{
    #if DEBUG_TRACE
        printf("%d :- end_resp_to_client()\n",  getpid());
    #endif

    if (client_fd != -1) {
        (void)close(client_fd);
```

```
            client_fd = -1;
        }
    }
```

## Client-Side Functions

Complementing the server are the client functions in `pipe_imp.c`. They are very similar to the server-side functions, except for the worryingly named `send_mess_to_server` function.

### Client Functions

1. After checking that a server is accessible, the `client_starting` function initializes the client-side pipe:

```
int client_starting(void)
{
    #if DEBUG_TRACE
        printf("%d :- client_starting\n",  getpid());
    #endif

    mypid = getpid();
    if ((server_fd = open(SERVER_PIPE, O_WRONLY)) == -1) {
        fprintf(stderr, "Server not running\n");
        return(0);
    }

    (void)sprintf(client_pipe_name, CLIENT_PIPE, mypid);
    (void)unlink(client_pipe_name);
    if (mkfifo(client_pipe_name, 0777) == -1) {
        fprintf(stderr, "Unable to create client pipe %s\n",
                   client_pipe_name);
        return(0);
    }
    return(1);
}
```

2. The `client_ending` function closes file descriptors and deletes the now-redundant named pipe:

```
void client_ending(void)
{
    #if DEBUG_TRACE
        printf("%d :- client_ending()\n",  getpid());
    #endif

    if (client_write_fd != -1) (void)close(client_write_fd);
    if (client_fd != -1) (void)close(client_fd);
    if (server_fd != -1) (void)close(server_fd);
    (void)unlink(client_pipe_name);
}
```

3. The `send_mess_to_server` function passes the request through the server pipe:

```
int send_mess_to_server(message_db_t mess_to_send)
{
    int write_bytes;

    #if DEBUG_TRACE
        printf("%d :- send_mess_to_server()\n",  getpid());
    #endif

    if (server_fd == -1) return(0);
    mess_to_send.client_pid = mypid;
    write_bytes = write(server_fd, &mess_to_send, sizeof(mess_to_send));
    if (write_bytes != sizeof(mess_to_send)) return(0);
    return(1);
}
```

As with the server-side functions you saw earlier, the client gets results back from the server using three functions, to cater to multiple search results.

## Getting Server Results

1. This client function starts to listen for the server response. It opens a client pipe as read-only and then reopens this pipe's file as write-only. You'll see why a bit later in the section.

```
    int start_resp_from_server(void)
    {
        #if DEBUG_TRACE
            printf("%d :- start_resp_from_server()\n",  getpid());
        #endif

        if (client_pipe_name[0] == '\0') return(0);
        if (client_fd != -1) return(1);

        client_fd = open(client_pipe_name, O_RDONLY);
        if (client_fd != -1) {
            client_write_fd = open(client_pipe_name, O_WRONLY);
            if (client_write_fd != -1) return(1);
            (void)close(client_fd);
            client_fd = -1;
        }
        return(0);
    }
```

2. Here's the main `read` from the server that gets the matching database entries:

```
    int read_resp_from_server(message_db_t *rec_ptr)
    {
        int read_bytes;
        int return_code = 0;

        #if DEBUG_TRACE
            printf("%d :- read_resp_from_server()\n",  getpid());
        #endif

        if (!rec_ptr) return(0);
        if (client_fd == -1) return(0);

        read_bytes = read(client_fd, rec_ptr, sizeof(*rec_ptr));
        if (read_bytes == sizeof(*rec_ptr)) return_code = 1;
        return(return_code);
    }
```

3. And finally, here's the client function that marks the end of the server response:

```
    void end_resp_from_server(void)
    {
        #if DEBUG_TRACE
            printf("%d :- end_resp_from_server()\n",  getpid());
        #endif

        /* This function is empty in the pipe implementation */
    }
```

The second, additional `open` of the client pipe for writing in `start_resp_from_server`,

```
    client_write_fd = open(client_pipe_name, O_WRONLY);
```

is used to prevent a race condition when the server needs to respond to several requests from the client in quick succession.

To explain this a little more, consider the following sequence of events:

1. The client writes a request to the server.
2. The server reads the request, opens the client pipe, and sends the response back, but is suspended before it gets as far as closing the client pipe.
3. The client opens its pipe for reading, reads the first response, and closes its pipe.
4. The client then sends a new command and opens the client pipe for reading.
5. The server then resumes running, closing its end of the client pipe.

Unfortunately, at this point the client is trying to read the pipe, looking for a response to its next request, but the `read` will return with 0 bytes because no process has the client pipe open for writing.

By allowing the client to open its pipe for both reading and writing, thus removing the need for repeatedly reopening the pipe, you avoid this race condition. Note that the client never writes to the pipe, so there's no danger of reading erroneous data.

## Application Summary

You've now separated the CD database application into a client and a server, enabling you to develop the user interface and the underlying database technology independently. You can see that a well-defined database interface allows each major element of the application to make the best use of computer resources. If you took things a little further, you could change the pipes implementation to a networked one and use a dedicated database server machine. You learn more about networking in Chapter 15.