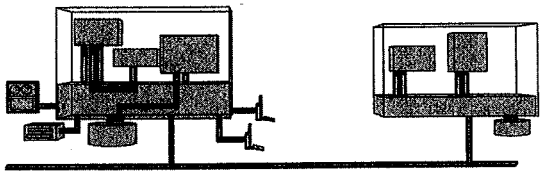
CHAPTER 10

I/O Redirection and Pipes



OBJECTIVES

Ideas and Skills

- I/O Redirection: What and why?
- Definitions of standard input, output, and error
- Redirecting standard I/O to files
- Using `fork` to redirect I/O for other programs
- Pipes
- Using `fork` with pipes

System Calls and Functions

- dup, dup2
- `pipe`

10.1 SHELL PROGRAMMING

How do the commands

```
ls > my.files
who | sort > userlist
```

work? How does the shell tell a program to send its output to a file instead of the screen? How does the shell connect the output stream of one process to the input stream of another process? What does the term *standard input* really mean?

In this chapter, we focus on a particular form of interprocess communication: *input/output (I/O) redirection* and *pipes*. We start by seeing how I/O redirection and

pipes help in writing shell scripts. We then look at underlying features of the operating system that make I/O redirection work. Finally, we write our own programs that change input and output streams for processes.

10.2 A SHELL APPLICATION: WATCH FOR USERS

Consider the following problem: You have a list of pals that use the same Unix machine you do. You want a program that notifies you when people log in or log out of the system so you can watch for your pals.

You *could* write a C program that uses the `utmp` file and interval timers. The program would open the `utmp` file, make a list of users, and then sleep for a while, rescan the `utmp` file, and report any changes. How much time and how much code would that take?

A simpler solution is to write a shell script. Unix already has a program that lists current users: `who`. Unix also includes programs to sleep and to process lists of strings. Here is a Unix script that reports all logins and logouts:

```
logic                                    shell code
---------------------------------        ----------------------
get list of users (call it prev)         who | sort > prev
while true                               while true ; do
  sleep                                     sleep 60
  get list of users (call it curr)          who | sort > curr
  compare lists                             echo "logged out:"
    in prev, not in curr -> logout          comm -23 prev curr
                                            echo "logged in:"
    in curr, not in prev -> login           comm -13 prev curr
  make prev = curr                          mv curr prev
repeat                                   done
```

In this script, we combine seven Unix tools, one `while` loop, and a generous helping of I/O redirection to build a program that solves the problem. Let us look at the details of the programs and the connections among these programs.

The first line in the script builds a list, sorted by username, of all users logged in when the script starts running. The `who` command outputs a list of users, and the `sort` command reads a list as input and outputs a sorted version of that list.

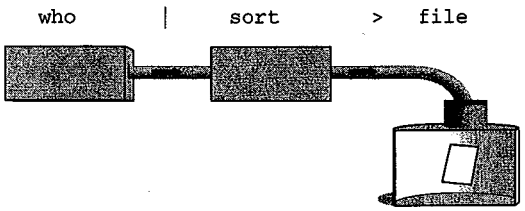


FIGURE 10.1
Connecting output of `who` to input of `sort`.

The line `who | sort > prev` tells the shell to run the commands `who` and `sort` at the same time, and to send the output of `who` directly to the input to `sort`. (See Figure 10.1.) The `who` command does not have to finish analyzing the `utmp` file before `sort` begins reading and sorting input. The two processes are scheduled to run in

small time slices, sharing CPU time with other processes on the system. Furthermore, the `sort > prev` part of the line tells the shell to send the output of `sort` into a file called `prev`, creating the file if it does not exist and replacing its contents if it does.

After sleeping for a minute, the script creates a new list of users in the file called `curr`. How can we compare two sorted lists of log-in records? The Unix tool `comm`, depicted in Figure 10.2, finds lines common to two sorted files. Given two files, there are three subsets: lines in set 1 only, lines in set 2 only, and lines in both sets. The `comm` command compares two sorted lists and prints out three columns, one for each of these subsets. Command-line options allow you to suppress any of the columns. For example, the two commands

```
comm -23 prev curr     # drop columns 2 and 3 => show lines only in prev
```

and

```
comm -13 prev curr     # drop columns 1 and 3 => show lines only in curr
```

produce exactly the two sets we want: those log-in records in the previous list, but not in the current list (logouts), and those log-in records not in the previous list, but only in the current list (logins).

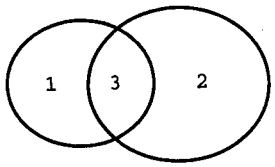


FIGURE 10.2

`comm` compares two lists and outputs three sets.

Finally, the command `mv curr prev` replaces the list called `prev` with the list called `curr`.

Lessons

This `watch.sh` script demonstrates three important ideas:

(a) Power of shell scripts—easier and quicker than C
(b) Flexibility of software tools—each tool does one specific, general task
(c) Use and value of I/O redirection and pipes

`watch.sh` shows how to use the > operator to treat files as variables of arbitrary size and structure. In the same way one writes

```
x = func_a(func_b(y));     /* store output of func_a of func_b in x */
```

in C, one writes

```
prog_b | prog_a > x        # store output of combination in x
```

in sh.

Questions

How does all this work? What role does the shell play in connecting processes? What role does the kernel play? What role do the individual programs play?

10.3 FACTS ABOUT STANDARD I/O AND REDIRECTION

All Unix I/O redirection is based on the principle of standard streams of data. Consider the `sort` tool. `sort` reads bytes from one stream of data, writes the sorted results to another stream, and reports any errors to a third stream. Ignoring for now the question of where these standard streams of data go, the `sort` utility has the basic shape shown in Figure 10.3. The three channels for data flow are as follows:

standard input—the stream of data to process
standard output—the stream of result data
standard error—a stream of error messages

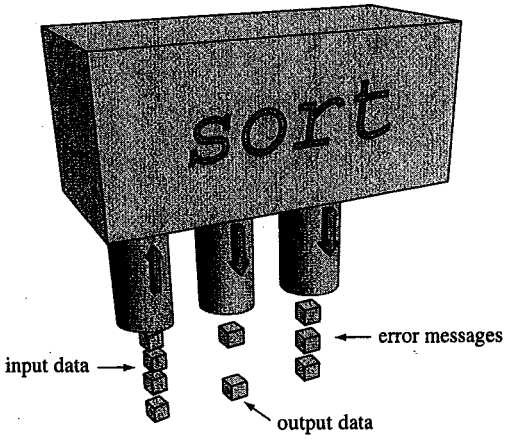


FIGURE 10.3
A software tool reads input and writes output and errors.

10.3.1 Fact One: Three Standard File Descriptors

All Unix tools use the three-stream model shown in Figure 10.3. The model is implemented via a simple rule. Each of these three streams is a specific file descriptor. Figure 10.4 shows the details.

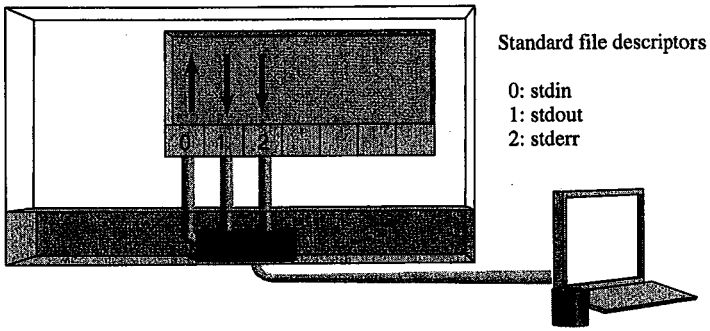


FIGURE 10.4
Three special file descriptors.

FACT: All Unix tools use file descriptors 0, 1, and 2.

Standard input *means* file descriptor 0, standard output *means* file descriptor 1, and standard error *means* file descriptor 2. Unix tools expect to find file descriptors 0, 1, and 2 already open for reading, writing, and writing, respectively.

10.3.2 Default Connections: the `tty`

When you run a Unix tool from the command line of the shell, `stdin`, `stdout`, and `stderr` are usually connected to your terminal. Therefore, the tool reads from the keyboard and writes output and error messages to the screen. For example, if you type `sort` and press the Enter key, your terminal will be connected to the `sort` tool. Type as many lines of input as you like. When you indicate end of file by pressing Ctrl-D on a line by itself, the `sort` program sorts the input and writes the result to `stdout`.

Most Unix tools process data from files or from standard input. If the tool is given file names on the command line, it reads input from those files. If there are no files named on the command line, the program reads from standard input.

10.3.3 Output Goes Only to `stdout`

On the other hand, most programs do not accept names for output files; they always write results to file descriptor 1 and errors to file descriptor 2.[1] If you want to send the output of a process to a file or to the input of another process, you change where the file descriptor goes.

10.3.4 The Shell, Not the Program, Redirects I/O

You tell the shell to attach file descriptor 1 to a file by using the output redirection notation: `cmd > filename`. The shell connects that file descriptor to the named file.

The program continues to write to file descriptor 1, unaware of the new data destination. The following program, called `listargs.c`, shows that the program does not even see the redirection notation on the command line:

```
/* listargs.c
 *              print the number of command line args, list the args,
 *              then print a message to stderr
 */
#include        <stdio.h>

main( int ac, char *av[] )
{
        int     i;

        printf("Number of args: %d, Args are:\n", ac);
```

[1]The commands `sort` and `dd` allow `stdout` overrides, but they have good reasons.

```
        for(i=0;i<ac;i++)
                printf("args[%d] %s\n", i, av[i]);

        fprintf(stderr,"This message is sent to stderr.\n");
}
```

`listargs` prints to standard output the list of command-line arguments. Notice that `listargs` does not print the redirection symbol and filename:

```
$ cc listargs.c -o listargs
$ ./listargs testing one two
args[0] ./listargs
args[1] testing
args[2] one
args[3] two
This message is sent to stderr.
$ ./listargs testing one two > xyz
This message is sent to stderr.
$ cat xyz
args[0] ./listargs
args[1] testing
args[2] one
args[3] two
$ ./listargs testing >xyz one two 2> oops
$ cat xyz
args[0] ./listargs
args[1] testing
args[2] one
args[3] two
$ cat oops
This message is sent to stderr.
```

These examples demonstrate some important facts about output redirection in the shell. The most important fact is that the shell does not pass the redirection symbol and filename to the command.

The second fact is that the redirection request can appear *anywhere* in the command and does not require spaces around the redirection symbol. Even a command like `> listing ls` is acceptable. Thus, the > sign does not terminate the command and arguments; it is just an added request.

The final fact is that many shells provide notation for redirecting other file descriptors. For example, `2>filename` redirects file descriptor 2, that is, standard error, to the named file.

10.3.5 Understanding I/O Redirection

We saw in `watch.sh` that I/O redirection is an integral part of Unix programming. We saw in `listargs.c` that the shell, not the tool, redirects input and output.

But *how* does the shell do I/O redirection? How can we write programs that redirect I/O? Our project for this chapter is to write programs that do three basic redirection operations:

who > userlist	attach `stdout` to a file
sort < data	attach `stdin` to a file
who | sort	attach `stdout` to stdin

10.3.6 Fact Two: The "Lowest-Available-fd" Principle

What is a file descriptor anyway? A file descriptor is a remarkably simple concept: It is an array index. Each process has a collection of files it has open. Those open files are kept in an array. A file descriptor is simply an index of an item in that array. Figure 10.5 illustrates the "lowest-available-file-descriptor" rule.

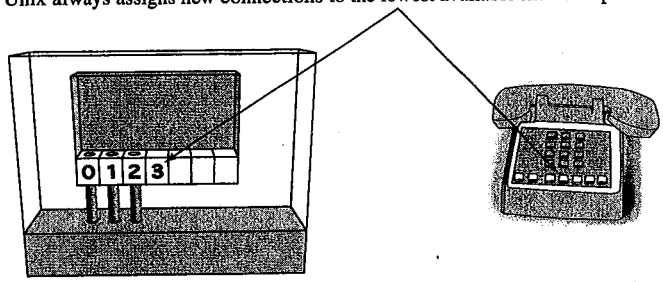


FIGURE 10.5
The "lowest-available-file-descriptor" rule.

FACT: When you open a file, you *always get* the lowest available spot in the array.

Making a new connection with file descriptors is like receiving a connection on a multiline phone. Callers dial a main number, and the internal phone system assigns each new connection an internal line. On many such systems, the next incoming call is assigned the *lowest available line*.

10.3.7 The Synthesis

We now have two basic facts. First, we have the convention that all Unix processes use file descriptors 0, 1, and 2 for the standard input, output, and error channels. Second, we have the fact that the kernel assigns the lowest available file descriptor when a process requests a new file descriptor. By combining these two facts, we can understand how I/O redirection works, and we can write programs that perform I/O redirection.

10.4 HOW TO ATTACH `stdin` TO A FILE

We now examine in detail how a program redirects standard input so that data come from a file. To be precise, processes do not read from files; processes read from file descriptors. If we attach file descriptor 0 to a file, that file becomes the source for standard input.

We examine three methods for attaching standard input to a file. Some of these methods are not appropriate for files, but are essential when we work with pipes.

10.4.1 Method 1: Close Then Open

The first method is the *close-then-open* technique. This technique is like hanging up to free a particular line and then picking up the telephone so you get that line. Here are the steps:

Starting, we have a typical configuration. The three standard streams are connected to the terminal driver. Data flow in through file descriptor 0 and data flow out through file descriptors 1 and 2. (See Figure 10.6.)

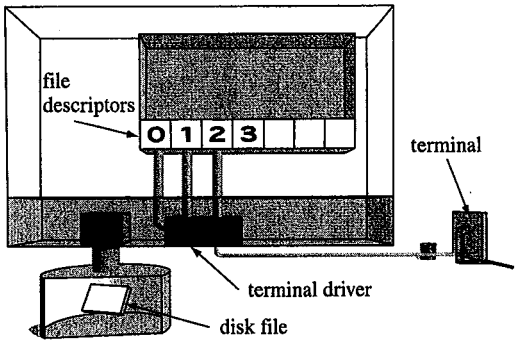


FIGURE 10.6
Typical starting configuration.

Then, close(0), the first step, is to hang up the connection to standard input. We call `close(0)` to break the connection from standard input to the terminal driver. Figure 10.7 shows how the first element in the array of file descriptors is now unused.

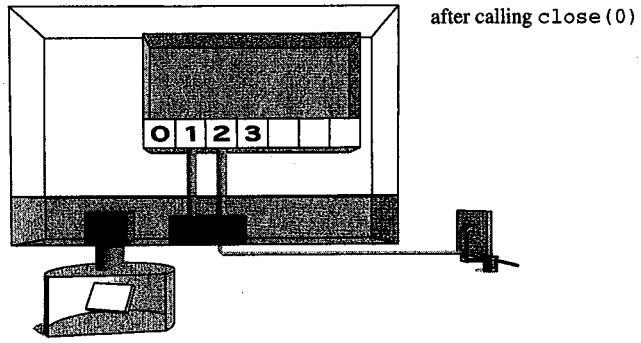


FIGURE 10.7
`stdin` is now closed.

Finally, open(filename,O_RDONLY), the last step, is to open the file you want to attach to `stdin`. The lowest number available file descriptor is 0, so the file you open will be attached at standard input. (See Figure 10.8.) Any functions that read from standard input will read from that file.

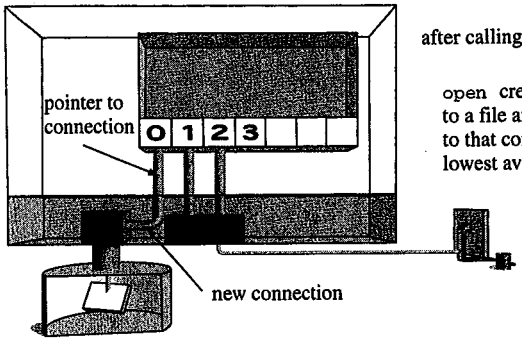


FIGURE 10.8
`stdin` now attached to file.

The following program uses the *close-then-open* method:

```
/* stdinredir1.c
 *      purpose: show how to redirect standard input by replacing file
 *               descriptor 0 with a connection to a file.
 *       action: reads three lines from standard input, then
 *               closes fd 0, opens a disk file, then reads in
 *               three more lines from standard input
 */
#include        <stdio.h>
#include        <fcntl.h>

main()
{
        int     fd ;
        char    line[100];

        /* read and print three lines */

        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );

        /* redirect input */

        close(0);
        fd = open("/etc/passwd", O_RDONLY);
        if ( fd != 0 ){
                fprintf(stderr,"Could not open data as fd 0\n");
                exit(1);
        }
```

```
        /* read and print three lines */

        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );
}
```

`stdinreader1` reads and prints three lines from standard input, redirects standard input, and then reads and prints three more lines from standard input. `stdinreader1` reads the first three lines from the keyboard, and then reads the next three lines from the `passwd` file:

```
$ ./stdinredir1
line1
line1
testing line2
testing line2
line 3 here
line 3 here
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
$
```

Nothing else special is going on. Just hang up the line and dial the new number. When the connection is made, you are now hearing from a new source of standard input.

10.4.2 Method 2: open..close..dup..close

Consider this situation: the telephone rings, you pick up the upstairs extension, but you realize you want to take the call on the downstairs phone. You tell someone downstairs to pick up that phone, giving you two connections to the caller, then you hang up the upstairs phone, leaving the only active connection on the downstairs phone. Does that sound familiar? The idea in this method is to duplicate the connection from the upstairs phone to the downstairs phone so you can hang up the upstairs phone without losing the connection.

The Unix system call `dup`, depicted in Figure 10.9, makes a second connection to an existing file descriptor. The method requires four steps:

open(file) — The first step is to open the file to which `stdin` should be attached. This call returns a file descriptor, which is not 0, since 0 is currently open.

close(0) — The next step is to close file descriptor 0. File descriptor 0 is now unused.

dup(fd) — The `dup(fd)` system call makes a duplicate of `fd`. The duplicate uses the *lowest number unused file descriptor*. Therefore, the duplicate of the connection to the file is at spot 0 in the array of open files. We have thereby attached the disk file to file descriptor 0.

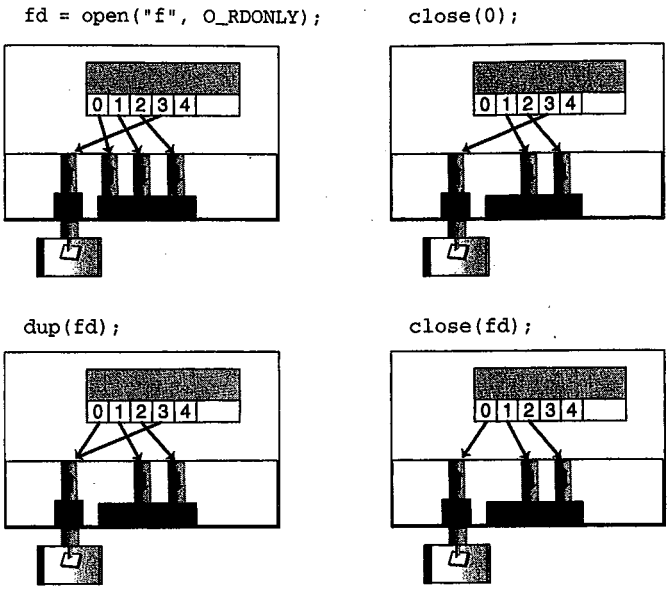


FIGURE 10.9
Using dup to redirect.

close(fd) Finally, we `close(fd)`, the original connection to the file, leaving only the connection on file descriptor 0. Compare this method to the technique of moving a phone call from one extension to another.

This program, `stdinredir2.c`, uses method 2:

```
/* stdinredir2.c
 *      shows two more methods for redirecting standard input
 *      use #define to set one or the other
 */
#include        <stdio.h>
#include        <fcntl.h>

/* #define      CLOSE_DUP               /* open, close, dup, close */
/* #define      USE_DUP2        /* open, dup2, close */

main()
{
        int     fd ;
        int     newfd;
        char    line[100];

        /* read and print three lines */

        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );

        /* redirect input */
```

```
        fd = open("data", O_RDONLY);     /* open the disk file   */
#ifdef CLOSE_DUP
        close(0);
        newfd = dup(fd);                 /* copy open fd to 0    */
#else
        newfd = dup2(fd,0);              /* close 0, dup fd to 0 */
#endif
        if ( newfd != 0 ){
                fprintf(stderr,"Could not duplicate fd to 0\n");
                exit(1);
        }
        close(fd);                       /* close original fd    */

        /* read and print three lines */

        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );
        fgets( line, 100, stdin ); printf("%s", line );
}
```

This four-step method is included for the pedagogical purpose of demonstrating the dup system call, an essential tool when working with pipes. A simpler method combines the `close(0)` and `dup(fd)` steps into a single system call, dup2.

10.4.3 System Call Summary: dup

dup, dup2	
PURPOSE	Copy a file descriptor
INCLUDE	`#include <unistd.h>`
USAGE	`newfd = dup(oldfd);` `newfd = dup2(oldfd, newfd);`
ARGS	`oldfd` file descriptor to copy `newfd` copy of oldfd
RETURNS	`-1` if error `newfd` new file descriptor

dup creates a copy of file descriptor *oldfd*. dup2 makes file descriptor *newfd* the copy of *oldfd*. The two file descriptors refer to the same open file. Both calls return the new file descriptor or −1 on error.

10.4.4 Method 3: open..dup2..close

The code for `stdinredir2.c` includes `#ifdef`-ed code to replace the `close(0)` and `dup(fd)` system calls with `dup2(fd,0)`. `dup2(orig,new)` makes a duplicate of file

descriptor *old* at file descriptor *new*, even if it has to close an existing connection on *new* first.

10.4.5 But the Shell Redirects `stdin` for Other Programs

These samples show how a program can attach its standard input to a file. In practice, of course, if a program wants to read a file, it can just open the file directly rather than changing standard input. The real value of these samples is to show how one program can change standard input for another program.

10.5 REDIRECTING I/O FOR ANOTHER PROGRAM: `who > userlist`

When a user types `who > userlist`, the shell runs the command `who` with the standard output of `who` attached to the file called `userlist`. How does that work?

The secret is the split second between `fork` and `exec`. After `fork`, the child process is still running the shell code, but is about to call `exec`. `exec` will replace the program running in the process, but it will not change either the attributes or the connections of the process. That is, after `exec`, the process will have the same user ID it had before, the process will have the same priority it had before, and the process *will have the same file descriptors it had before* the `exec`. To repeat, a program gets the open files of the process into which it is loaded. Figure 10.10 illustrates the redirection of output for a child.

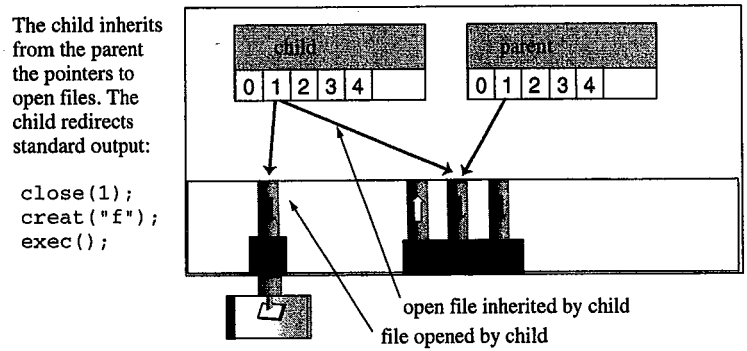


FIGURE 10.10

The shell redirects output for a child.

Let us watch a process use this principle to redirect standard output:

1. *Start here*

   In Figure 10.11, a process is running in user space. File descriptor 1 is attached to an open file called *f* as shown. To make the picture clearer, other open files are not shown.

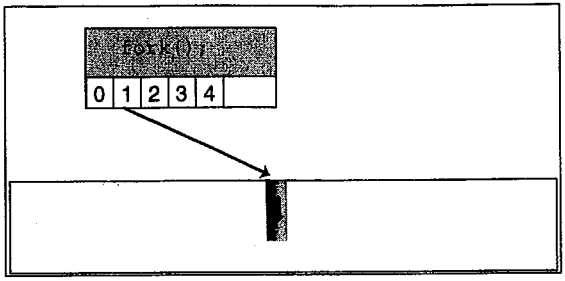


FIGURE 10.11
A process about to fork and its standard output.

2. *After parent calls fork*

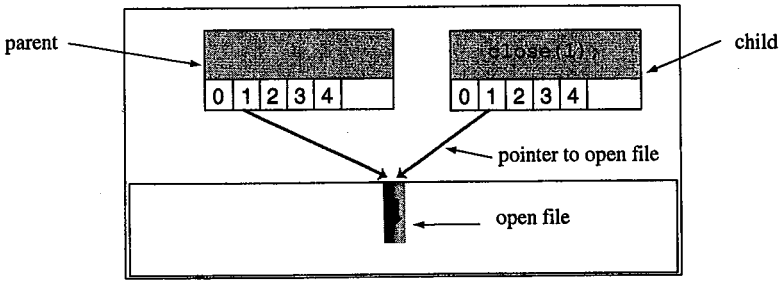


FIGURE 10.12
Standard output of child is copied from parent.

In Figure 10.12, a new process appears. This process runs the same code as the original process, but knows it is a child process. The child process contains the same code, the same data, and the same set of open files as its parent. Therefore the item in spot 1 in the array of open files refers, also, to file *f*. The child calls `close(1)`.

3. *After child calls* `close(1)`

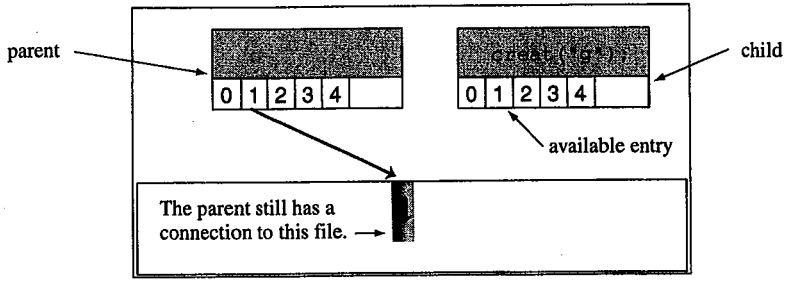


FIGURE 10.13
The child can close its standard output.

In Figure 10.13, the parent process has not called `close(1)`, so file descriptor 1 in the parent still connects to file *f*. In the child process, file descriptor 1 is the lowest unused file descriptor. The child now opens a file called `g`.

4. *After child calls* `creat("g",m)`

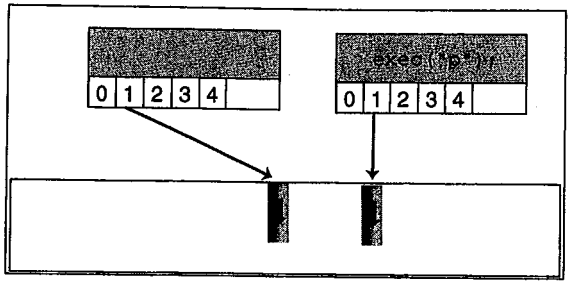


FIGURE 10.14

Child opens a new file, getting `fd = 1`.

In Figure 10.14, file descriptor 1 is now attached to `g`. Standard output in the child is redirected to `g`. The child now calls `exec` to run `who`.

5. *After child* `exec`*s a new program*

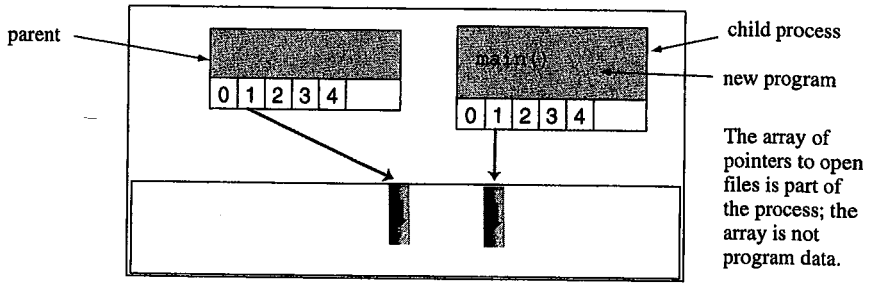


FIGURE 10.15

Child runs a program with new standard output.

In Figure 10.15, the child executes the `who` program. The code and data for the shell are removed from the child process and are replaced by the code and data for the `who` program. The *file descriptors* are retained across the `exec`. Open files are not part of the code nor data of a program; they are attributes of a process.

The `who` command writes the list of users to file descriptor 1. Unbeknownst to `who`, that stream of output bytes flows into file `g`.

The program `whotofile.c` demonstrates the method:

```
/* whotofile.c
 *      purpose: show how to redirect output for another program
 *         idea: fork, then in the child, redirect output, then exec
 */
```

```
#include        <stdio.h>

main()
{
        int     pid ;
        int     fd;

        printf("About to run who into a file\n");

        /* create a new process or quit */
        if( (pid = fork() ) == -1 ){
                perror("fork"); exit(1);
        }
        /* child does the work */
        if ( pid == 0 ){
                close(1);                               /* close, */
                fd = creat( "userlist", 0644 );         /* then open */
                execlp( "who", "who", NULL );           /* and run     */
                perror("execlp");
                exit(1);
        }
        /* parent waits then reports */
        if ( pid != 0 ){
                wait(NULL);
                printf("Done running who.  results in userlist\n");
        }
}
```

10.5.1 Summary of Redirection to Files

Three basic facts make it easy under Unix to attach standard input, standard output, and standard error to disk files:

(a) Standard input, output, and error are file descriptors 0, 1, and 2.
(b) The kernel always uses the lowest numbered unused file descriptor.
(c) The set of file descriptors is passed unchanged across exec calls.

The shell uses the interval in the child between fork and exec to attach standard data streams to files.

The shell also supports notation of the following form:

```
who >> userlog
sort < data
```

Writing the code to perform these two operations is left as an exercise.

10.6 PROGRAMMING PIPES

We saw how to write a program that attaches standard output to a file. We now examine how to use pipes to connect the standard output of one process to the standard input of another process. Figure 10.16 shows how pipes work. A pipe is a one-way data

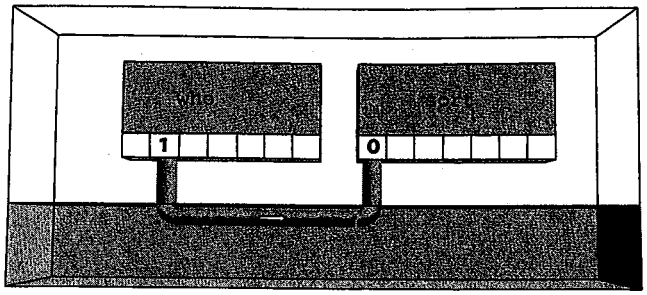


FIGURE 10.16
Two processes connected by a pipe.

channel in the kernel. A pipe has a reading end and a writing end. To write `who | sort`, we need two skills: how to create a pipe and how to connect standard input and output to a pipe.

10.6.1 Creating a Pipe

A pipe is illustrated in Figure 10.17. Use the `pipe` system call, summarized as follows, to create a pipe:

pipe	
PURPOSE	Create a pipe
INCLUDE	`#include <unistd.h>`
USAGE	`result = pipe(int array[2])`
ARGS	`array` an array of two ints
RETURNS	`-1` if error `0` if success

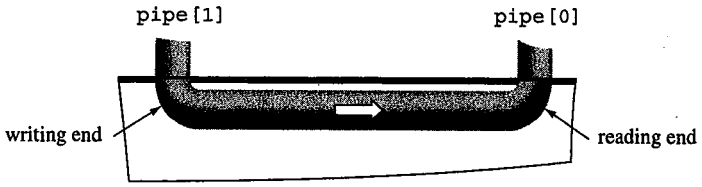


FIGURE 10.17
A pipe.

`pipe` creates the pipe and connects its two ends to two file descriptors. `array[0]` is the file descriptor of the reading end, and `array[1]` is the file descriptor of the writing end. The internals of the pipe, like the internals of an open file, are hidden in the kernel. The process sees two file descriptors.

The pair of before and after shots in Figure 10.18 shows a process creating a pipe. The *before* shot shows the standard set of file descriptors. The *after* shot shows the newly created pipe in the kernel and the two connections to that pipe in the process. Notice that `pipe`, like `open`, uses the lowest-numbered available file descriptors.

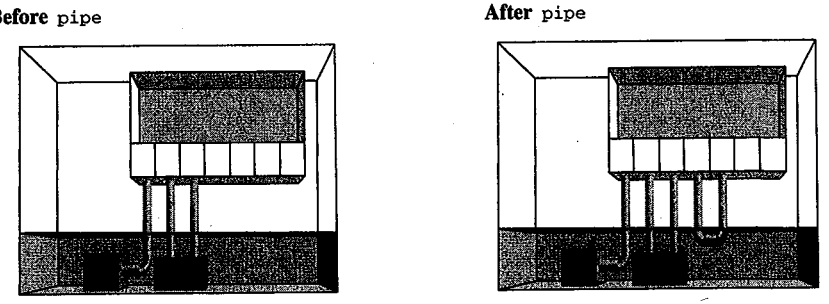


FIGURE 10.18

A process creates a pipe.

The program, `pipedemo.c`, creates a pipe and then uses the pipe to send data to itself:

```
/*  pipedemo.c  * Demonstrates: how to create and use a pipe
 *              * Effect: creates a pipe, writes into writing
 *                end, then runs around and reads from reading
 *                end.  A little weird, but demonstrates the idea.
 */
#include        <stdio.h>
#include        <unistd.h>

main()
{
        int     len, i, apipe[2];       /* two file descriptors */
        char    buf[BUFSIZ];            /* for reading end      */

        /* get a pipe */
        if ( pipe ( apipe ) == -1 ){
                perror("could not make pipe");
                exit(1);
        }
        printf("Got a pipe! It is file descriptors: { %d %d }\n",
                                                apipe[0], apipe[1]);

        /* read from stdin, write into pipe, read from pipe, print */
```

```
        while ( fgets(buf, BUFSIZ, stdin) ){
                len = strlen( buf );
                if ( write( apipe[1], buf, len) != len ){       /* send */
                        perror("writing to pipe");              /* down */
                        break;                                  /* pipe */
                }
                for ( i = 0 ; i<len ; i++ )                     /* wipe */
                        buf[i] = 'X' ;
                len = read( apipe[0], buf, BUFSIZ ) ;           /* read */
                if ( len == -1 ){                               /* from */
                        perror("reading from pipe");            /* pipe */
                        break;
                }
                if ( write( 1 , buf, len ) != len ){            /* send  */
                        perror("writing to stdout");            /* to    */
                        break;                                  /* stdout */
                }
        }
}
```

Figure 10.19 depicts the flow of bytes from keyboard to process, from process to pipe, from pipe to process, and from process back to terminal.

We now know how to create a pipe and how to write data into a pipe and how to read data from a pipe. In practice, one rarely writes a program that sends data to itself. By combining `pipe` with `fork`, though, we can connect two processes.

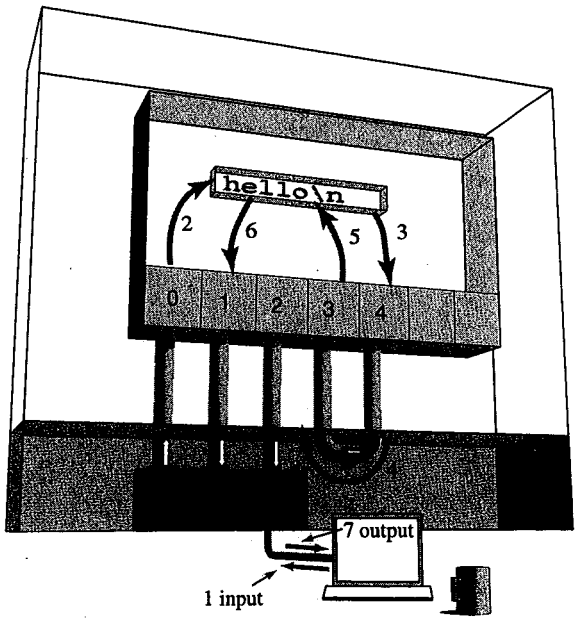


FIGURE 10.19
Data flow in `pipedemo.c`.

10.6.2 Using `fork` to Share a Pipe

When a process creates a pipe, the process has connections to both ends of the pipe. When that process calls `fork`, the child process, a copy of the parent, also has connections to the pipe, as shown in Figure 10.20. Parent and child can write bytes to the writing end of the pipe, and parent and child can read bytes from the reading end of the pipe. (See Figure 10.21.) Both processes can read and write, but a pipe is most effective when one process writes data and the other process reads the data.

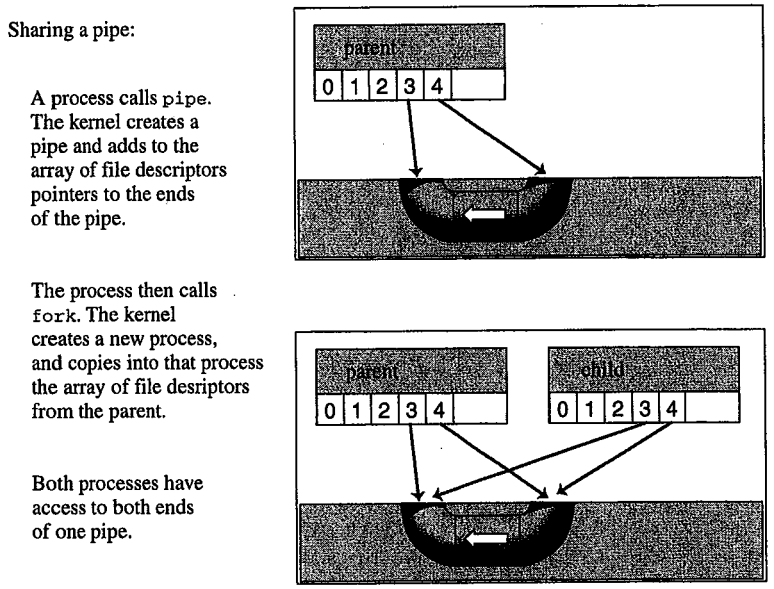


FIGURE 10.20
Sharing a pipe.

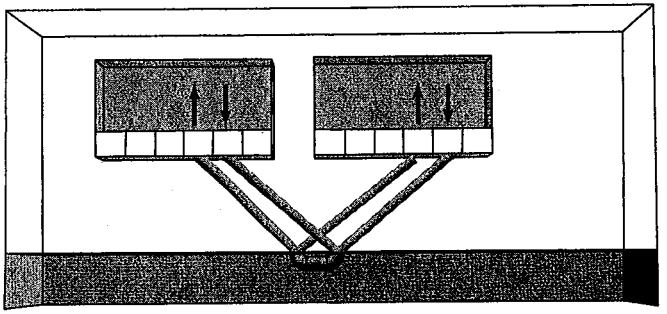


FIGURE 10.21
Interprocess data flow.

This program, `pipedemo2.c`, shows how to combine `pipe` and `fork` to create a pair of processes that communicate through a pipe:

```
/* pipedemo2.c  * Demonstrates how pipe is duplicated in fork()
 *              * Parent continues to write and read pipe,
 *                but child also writes to the pipe
 */
#include        <stdio.h>

#define CHILD_MESS      "I want a cookie\n"
#define PAR_MESS        "testing..\n"
#define oops(m,x)       { perror(m); exit(x); }

main()
{
        int     pipefd[2];              /* the pipe     */
        int     len;                    /* for write    */
        char    buf[BUFSIZ];            /* for read     */
        int     read_len;

        if ( pipe( pipefd ) == -1 )
                oops("cannot get a pipe", 1);

        switch( fork() ){
                case -1:
                        oops("cannot fork", 2);

                /* child writes to pipe every 5 seconds */
                case 0:
                        len = strlen(CHILD_MESS);
                        while ( 1 ){
                              if (write(pipefd[1], CHILD_MESS, len) != len )
                                        oops("write", 3);
                               sleep(5);
                             }

                /* parent reads from pipe and also writes to pipe */
                default:
                        len = strlen( PAR_MESS );
                        while ( 1 ){
                              if ( write( pipefd[1], PAR_MESS, len)!=len )
                                        oops("write", 4);
                              sleep(1);
                              read_len = read( pipefd[0], buf, BUFSIZ );
                              if ( read_len <= 0 )
                                      break;
                              write( 1 , buf, read_len );
                             }
        }
}
```

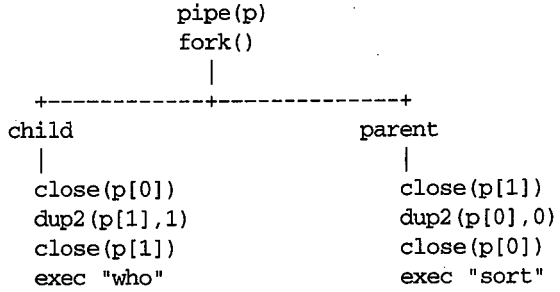
10.6.3 The Finale: Using `pipe`, `fork`, and `exec`

We now know all the ideas and skills required to write a program that connects the output of `who` to the input of `sort`. We know how to create a pipe, we know how to share a pipe between two processes, we know how to change the standard input of a process, and we know how to change the standard output of a process.

We combine all these skills to write a general-purpose program called `pipe` that takes the names of two programs as arguments. The examples

```
pipe who sort
pipe ls  head
```

show two uses of `pipe`. The logic of the program is as follows:



Here is the code:

```
/* pipe.c
 *      * Demonstrates how to create a pipeline from one process to another
 *      * Takes two args, each a command, and connects
 *        av[1]'s output to input of av[2]
 *      * usage: pipe command1 command2
 *        effect: command1 | command2
 *      * Limitations: commands do not take arguments
 *      * uses execlp() since known number of args
 *      * Note: exchange child and parent and watch fun
 */
#include        <stdio.h>
#include        <unistd.h>

#define oops(m,x)       { perror(m); exit(x); }

main(int ac, char **av)
{
        int     thepipe[2],                     /* two file descriptors */
                newfd,                          /* useful for pipes     */
```

```
        pid;                                    /* and the pid          */
    if ( ac != 3 ){
            fprintf(stderr, "usage: pipe cmd1 cmd2\n");
            exit(1);
    }
    if ( pipe( thepipe ) == -1 )            /* get a pipe           */
            oops("Cannot get a pipe", 1);
    /* ------------------------------------------------------------ */
    /*      now we have a pipe, now let's get two processes         */

    if ( (pid = fork()) == -1 )                     /* get a proc   */
            oops("Cannot fork", 2);

    /* ------------------------------------------------------------ */
    /*      Right Here, there are two processes                     */
    /*            parent will read from pipe                        */
    if ( pid > 0 ){                 /* parent will exec av[2]       */
            close(thepipe[1]);      /* parent doesn't write to pipe */

            if ( dup2(thepipe[0], 0) == -1 )
                    oops("could not redirect stdin",3);

            close(thepipe[0]);      /* stdin is duped, close pipe   */
            execlp( av[2], av[2], NULL);
            oops(av[2], 4);
     }

    /*      child execs av[1] and writes into pipe                  */
    close(thepipe[0]);              /* child doesn't read from pipe */

    if ( dup2(thepipe[1], 1) == -1 )
            oops("could not redirect stdout", 4);

    close(thepipe[1]);              /* stdout is duped, close pipe  */
    execlp( av[1], av[1], NULL);
    oops(av[1], 5);
}
```

The `pipe.c` program uses the same ideas and techniques the shell uses to create pipelines. The shell, though, does not run an external program like `pipe.c`. The shell has to create the pipe, then fork to create the two processes, then redirect standard input and output to the pipe, and finally exec the two programs.

10.6.4 Technical Details: Pipes Are Not Files

In many ways, pipes look like regular files. A process uses `write` to put data into a pipe and uses `read` to get data from a pipe. A pipe, like a file, appears as a sequence of bytes without any particular block or record structure. In other ways, pipes differ from files. What, for instance, does end of file mean for a pipe? The following technical details clarify some of these similarities and differences.

Reading from Pipes

1. `read` *on a pipe blocks*
   When a process tries to `read` from a pipe, the call blocks until some bytes are written into the pipe. What prevents a process from waiting forever?
2. *Reading EOF on a pipe*
   When all writers close the writing end of the pipe, attempts to `read` from the pipe return 0, that is, end of file.
3. *Multiple readers can cause trouble*
   A pipe is a queue. When a process `reads` bytes from a pipe, those bytes are no longer in the pipe. If two processes try to read from the same pipe, one process will get some of the bytes from the pipe, and the other process will get the other bytes. Unless the two processes use some method to coordinate their access to the pipe, the data they read are likely to be incomplete.

Writing to Pipes

4. `write` *to a pipe blocks until there is space*
   Pipes have a finite capacity that is far lower than the file-size limit imposed on disk files. When a process tries to `write` to a pipe, the call blocks until there is enough space in the pipe. If a process wants to write, say, 1000 bytes, and there is only room for 500 bytes, the call waits until 1000 bytes of space are available. What if the process wanted to write a million bytes? Would the call wait forever?
5. `write` *guarantees a minimum chunk size*
   The POSIX standard states that the kernel will not split up chunks of data into blocks smaller than 512 bytes. Linux guarantees an unbroken buffer size of 4096 for pipes. If two different processes write to a pipe, and each process limits its messages to 512 bytes, the processes are assured their messages will not be split.
6. `write` *fails if no readers*
   If all readers have closed the reading ends of a pipe, then an attempt to `write` to the pipe can lead to trouble. If data were accepted, where would they go? To avoid losing data, the kernel uses two methods to notify a process that `write` is futile. The kernel sends `SIGPIPE` to the process. If that kills the process, no further action is required. Otherwise, `write` returns -1 and sets `errno` to `EPIPE`.

SUMMARY

MAIN IDEAS

- Input/Output redirection allows separate programs to work as a team, each program a specialist.
- The Unix convention is that programs read input from file descriptor 0, write results to file descriptor 1, and report errors to file descriptor 2. Those three file descriptors are called standard input, standard output, and standard error.

- When you log in to Unix, the log-in procedure sets up file descriptors 0, 1, and 2. These connections, and all open file descriptors, are passed from parent to child and across the exec system call.
- System calls that create file descriptors always use the lowest-numbered free file descriptor.
- Redirecting standard input, output, or error means changing where file descriptors 0, 1, or 2 connect. There are several techniques for redirecting standard I/O.
- A pipe is a data queue in the kernel with each end attached to a file descriptor. A program creates a pipe with the `pipe` system call.
- Both ends of a pipe are copied to a child process when the parent calls `fork`.
- Pipes can only connect processes that share a common parent.

WHAT NEXT?

The traditional Unix pipe carries data between processes in one direction. What if two processes wanted to pass data back and forth? What if two processes were not related, or if two processes were on different computers? In the next several chapters, we look at pipes in more detail and then study network programming. The idea of a pipe generalizes to the idea of a socket.

EXPLORATIONS

10.1 *Meaning of* >> The >> notation tells the shell to append output to a file. Does the shell use auto-append mode (see Chapter 5), or does it simply seek to the end of the file and start writing there? Devise an experiment using shell scripts to answer the question.

10.2 In `pipe.c`, the parent process runs the program that consumes data, and the child process runs the program that produces data. What difference would it make if those roles were reversed? By changing the test `if ( pid > 0 )` to `if ( pid == 0 )`, the roles will be reversed. What happens? Why?

10.3 What changes do you need to make to your shell to include pipes? First, how would you modify the flow of control to identify and handle commands that end with a pipe sign? Second, what if there are several commands separated by pipe signs?

10.4 In `pipe.c`, the reading process, `sort`, closes its copy of the writing end of the pipe. Change the code so the reading process does *not* close the writing end of the pipe. Now run the program and explain its behavior.

10.5 *Adding > and < to your shell* We examined earlier in this chapter the notation to attach standard input or standard output to a file. We saw that the redirection symbol and filename may appear anywhere in the command line. We also saw that the symbol and filename are not part of the list of arguments passed to the new program.

Where in the logic of our small shell should we identify the request to change input or output to a disk file?

Where in the logic of our small shell should the redirection be performed?

What if a user typed `set > varlist` ? Does the shell allow you to redirect the output of built-in commands? How can you add that to our shell?

10.6 *Protecting users* What if a user types `sort <data >data` . What is the problem with this request? What do standard Unix shells do about it? How can your shell handle this problem?

10.7 We examined methods for attaching the standard input or standard output of a process to a file. All our examples have assumed regular plain disk files. Can I/O redirection work for files that represent devices? That is, what if you `close(0)` and `open("/dev/tty",0)`? What does the shell do with the command `who > /dev/tty`?

10.8 In `pipe.c`, we call `fork` and `exec`, but we do not call `wait`. Why not?

10.9 How is `dup` like `link`? Discuss pointers.

PROGRAMMING EXERCISES

10.10 Modify the `watch.sh` script so it has nicer features.

(a) This version prints out logins and logouts for all users. A more useful version would accept as a command-line argument a name of a file that contains a list of users to watch.

(b) This version prints out something each time through the loop, even if nothing has changed. Modify the program so it prints out the *new logins* and *new logouts* messages only if there is something to show.

(c) The `who` command lists in addition to username, the log-in time and the terminal name. That may be more information than you want. If a user connects using a second window, that may not be interesting to you. Write a version of the program that reports when a watched user changes from "is logged on" to "is not logged on," regardless of terminal.

(d) This version stores its data in files called `prev` and `curr` in the current directory and leaves those files there when the program stops running. This design is poor for several reasons. What are some of the reasons? Revise the script to use temporary files and to remove those files at exit. Read about the `trap` command in the shell. Examine the use of the `mktemp` command.

10.11 Modify the `whotofile.c` sample program so it appends the output of `who` to a file. Make sure your program works if the file does not exist.

10.12 Write a program called `sortfromfile.c` that redirects the input of the `sort` command so it reads from a file. The filename is specified on the command line.

10.13 Extend `pipe.c` to handle three-part pipelines. This new version should accept three program names as arguments and run them as a pipeline. The command

```
pipe3 who sort head
```

should have the same effect as `who | sort | head`.

10.14 Extend the `pipe3` program in the previous problem to handle an arbitrary number of arguments.

10.15 *process tee* The `tee` utility lets you redirect data to a file and also pass the data to the next command in a pipeline. For example,

```
who | tee userlist | sort > list2
```

produces an unsorted file and a sorted file: `userlist` and `list2`. The argument to `tee` is the name of a file; read the manual for more details. Write a program called `progtee` that redirects data to a program and also passes the data to the next command in a pipeline. For example, the pipeline

```
who | progtee mail smith | sort | progtee mail -s "hello" root > list2
```

sends an unsorted list to smith, sends a sorted list to root, and puts a copy of the sorted list into the file `list2`.

10.16 *isatty* Programs that write to standard output do not usually care if the file descriptor leads to a terminal or to a disk file. The text suggests that a process has no way of knowing where the file descriptor leads. This is false. The library function `isatty(fd)` returns a true value if *fd* refers to a terminal. `isatty` uses the system call `fstat`. Read about `fstat` and use that information to write a function `isaregfile` that returns a true value if its argument is a file descriptor connected to a regular file.