6 Basic Interprocess Communication

6.1 Introduction

Now that we know how to create processes, we want to connect them so they can communicate. We'll do this with pipes in this chapter, using basic techniques that have always worked on all versions of UNIX. In the next chapter we'll start exploring interprocess communication using techniques that are more efficient and robust, less universally available, and more troublesome to program.

Pipes are familiar to most UNIX users as a shell facility. For instance, to display a sorted list of who's logged in, you can type:

```
$ who | sort | more
```

There are three processes here, connected with two pipes. Data flows in one direction only, from `who` to `sort` to `more`. It's also possible to set up pipelines for two-way communication (from process A to B and from B back to A) and pipelines in a ring (from A to B to C to A) using system calls. Most shells, however, provide no notation for these more elaborate arrangements, so they are unknown to most UNIX users.[1]

I'll begin by showing some simple examples of processes connected for one-directional communication. Then we'll improve on the primitive shell we developed in Chapter 5. Our new shell will be complete enough to be called "real"—it will handle pipelines, background processes, I/O redirection, and quoted argu-

1. These arrangements can be set up with FIFOs, but the shell is an innocent bystander—it thinks they are regular files.

ments. It will lack file-name generation (e.g., `ls t*.?`) and programming constructs (e.g., `if` statements). At the end I'll show how to connect processes for two-way communication and expose the deadlock problems that can arise.

6.2 Pipes

This section is about unnamed pipes, although much of the behavior described applies to FIFOs (named pipes) as well. FIFOs are covered in detail in Section 7.2.

6.2.1 `pipe` System Call

pipe—create pipe

```
#include <unistd.h>

int pipe(
    int pfd[2]              /* file descriptors */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The `pipe` system call creates a pipe, which is a communication channel represented by two file descriptors that are returned in the `pfd` array. Writing to `pfd[1]` puts data in the pipe; reading from `pfd[0]` gets it out.

There's a parameter called `PIPE_BUF` in UNIX documentation that you can think of as the buffer size of the pipe. If several processes or threads are writing to the same pipe, a write of `PIPE_BUF` or fewer bytes is guaranteed to be atomic—there will be no interleaving of data from other writers. This property is extremely important if several processes are writing structured data, since without it there would be no guarantee that a reader could read valid data. `PIPE_BUF` is always at least 512, but you can get the actual value for a particular pipe at run-time with a call to `fpathconf` (Section 1.5.6):

```
int pfd[2];
long v;

ec_neg1( pipe(pfd) )
errno = 0;
v = fpathconf(pfd[0], _PC_PIPE_BUF);
```

```
if (v == -1)
    if (errno != 0)
         EC_FAIL
    else
         printf("No limit for PIPE_BUF\n");
else
    printf("PIPE_BUF = %ld\n", v);
```

On my systems, I got 5120 on Solaris, 512 on FreeBSD, and 4096 on Linux.

Initially the `O_NONBLOCK` (Section 4.2.2) is clear for a pipe; that is, reading and writing may block. As you would guess, you can set the flag with `fcntl` (Section 3.8.3). How this flag affects `read`s and `write`s is explained below.

6.2.2 Pipe (and FIFO) I/O Behavior

Everything in this section applies to both pipes and to FIFOs (discussed further in Section 7.2), and here the term "pipe" means both.

Some I/O system calls act differently on pipe file descriptors from the way they do on ordinary files, and some do nothing at all, as summarized by the following list (these are the main ones; [SUS2002] has all the details):

`write` Data written to a pipe is sequenced in order of arrival. Normally (`O_NONBLOCK` clear), if the pipe becomes full, `write` will block until enough old data is removed by `read`; there are no partial writes. The capacity of a pipe varies with the UNIX implementation, but obviously it is always at least `PIPE_BUF` bytes. If `O_NONBLOCK` is set and the amount to be written is `PIPE_BUF` or less, `write` will either write the data immediately or return –1 with `errno` set to `EAGAIN`; there are no partial writes. But if the amount is greater than `PIPE_BUF`, a partial write is possible.

`read` Data is read from a pipe in order of arrival, just as it was written. Once read, data can't be reread or put back. Normally (`O_NONBLOCK` clear), if the pipe is empty, `read` will block until at least one byte of data is available, unless all writing file descriptors are closed, in which case the `read` will return with a 0 count (the usual end-of-file indication). But the byte count given as the third argument to read will not necessarily be satisfied—only as many bytes as are present at that instant will be read, and an appropriate count will be returned. The byte count will never be exceeded, of course; unread bytes will remain for the next read. If `O_NONBLOCK` is set, a read on an empty pipe will return –1 with `errno` set to `EAGAIN`.

`close` Means more on a pipe than it does on a file. Not only does it free up the file descriptor for reuse, but when all writing file descriptors are closed it acts as an end-of-file for a reader. If all reading file descriptors are closed, a `write` on a writing file descriptor will cause an error. A fatal signal is also normally generated; see Section 9.1.3.

`fstat` Not very useful on pipes, except to determine that the file descriptor is open to a pipe. The size returned is usually the number of bytes in the pipe, but this is not required by any UNIX standard.

`dup` This system call and `dup2` are explained in Section 6.3.

`lseek` Not used with pipes. This means that if a pipe contains a sequence of messages, it isn't possible to look through them for the message to be read next. Like toothpaste in a tube, you have to get it out to examine it, and then there's no way to put it back. This is one reason why pipes are awkward for application programs that pass messages between processes.

System calls not explicitly listed (e.g., `select`, `poll`) operate on pipes just as you would expect. For example, when a file descriptor tested by `select` is open to a pipe, `select` tests whether a `read` or `write` would block.

For writes, the relationships between atomic vs. nonatomic, blocking vs. nonblocking, and complete vs. partial vs. deferred (–1 return with an `errno` of `EAGAIN`) is a little complicated. Table 6.1 (based on tables in POSIX1990) should help. Columns one and two contain the possibilities for the state of the `O_NONBLOCK` flag and the amount being written, and the last three columns indicate what happens for a full pipe, a pipe that can receive part of the data immediately, and a pipe that can receive all the data.

In the table, the notation "complete write" means that `write` doesn't return until the full requested amount is written. "Nonatomic" means that the data is all in the pipe, but not necessarily contiguously (not even the first `PIPE_BUF` bytes are guaranteed to be contiguous in this case). "`EAGAIN`" means a return from write of –1 with `errno` set to `EAGAIN`. "Partial" means that the value returned by `write` is less than the requested amount.

The reason for the notation "might block" at the end of the second row is that while it's true that when the `write` starts all the data will fit, since it is nonatomic, another process or thread could fill some of the pipe before the `write` completes, forcing it to block.

Table 6.1 Writing to a Pipe

O_NONBLOCK?	Amount to write	None immediately writable	Some immediately writable (>=1 and < amt.)	All immediately writable
clear	<=PIPE_BUF	blocks; complete write; atomic	blocks; complete write; atomic	does not block; complete write; atomic
clear	>PIPE_BUF	blocks; complete write; nonatomic	blocks; complete write; nonatomic	might block; complete write; nonatomic
set	<=PIPE_BUF	EAGAIN	EAGAIN	does not block; complete write; atomic
set	>PIPE_BUF	EAGAIN	does not block; partial or EAGAIN; nonatomic	does not block; complete, partial, or EAGAIN; nonatomic

Summarizing writes to a pipe one more time:

- If the requested amount is PIPE_BUF or less, writes are always atomic (which implies never partial).
- If O_NONBLOCK is clear (the normal case), writes are never partial, even when they are nonatomic.
- The only partial writes occur when O_NONBLOCK is set and the requested amount is greater than PIPE_BUF.

Table 6.2 is for reads.

Table 6.2 Reading from a Pipe

O_NONBLOCK?	None immediately readable	Some or all immediately readable (>=1 and <= amt.)
clear	blocks unless no writers (0 returned)	does not block; possibly partial read
set	EAGAIN unless no writers (0 returned)	does not block; possibly partial read

Note that the `read` table is much simpler than the `write` table, because there's never a guarantee of atomicity or of a complete read. `Read`s work like this:

- If all writing file descriptors are closed, a `read` on an empty pipe always immediately returns 0, which most programs treat as an end-of-file.
- If a writing file descriptor is open, a `read` on an empty pipe blocks or not depending on whether `O_NONBLOCK` is set.
- A `read` on a nonempty pipe always returns immediately, no matter what the relationship between the requested amount and the amount of data in the pipe. The amount actually read is returned.
- Because there is no guarantee of atomicity, you must never allow multiple readers unless you have another concurrency-control mechanism (e.g., an interprocess semaphore) to prevent simultaneous reading. (In practice, you will seldom do this—you'll use something like message queues instead.)

Here are additional guidelines to keep in mind:

- If you have one writer and one reader (e.g., a shell pipeline) and the reader is prepared for partial reads (e.g., uses Standard C I/O functions), write whatever amount you like, although multiples of the block size are most efficient, as we saw in Section 2.12.
- If you have multiple writers (and one reader), always write `PIPE_BUF` or less. There's no practical way to get pipes to work correctly with a greater amount, unless you use another synchronization mechanism, such as a semaphore (Section 7.8).
- Don't assume anything about the value of `PIPE_BUF` other than that it is at least 512. If you really need to know what it is, use `fpathconf`.
- Even though the standard doesn't require atomic reads, essentially all implementations make them atomic if the requested amount is `PIPE_BUF` or less. Use this fact at your own risk.
- Remember that to get an end-of-file (0 return from `read`) all writing file descriptors must be closed, including any in the process that's doing the reading. This point will be clearer later on, when we show how to connect two processes with a pipe.

6.2.3 Pipe Examples

(Now I'm talking once again only about *unnamed* pipes; FIFO examples are in the next chapter.)

Considering a single process only, of what use is a pipe? None, but such an example is very informative:

```
void pipetest(void)
{
    int pfd[2];
    ssize_t nread;
    char s[100];

    ec_neg1( pipe(pfd) )
    ec_neg1( write(pfd[1], "hello", 6) )
    ec_neg1( nread = read(pfd[0], s, sizeof(s)) )
    if (nread == 0)
        printf("EOF\n");
    else
        printf("read %ld bytes: %s\n", (long)nread, s);
    return;

EC_CLEANUP_BGN
    EC_FLUSH("pipetest");
EC_CLEANUP_END
}
```

The output:

```
read 6 bytes: hello
```

We could safely write 6 bytes into the pipe without worrying about it filling up, since that's well below 512 (the minimum for `PIPE_BUF`). But if we wrote much more and did fill the pipe, the `write` would block until the `read` emptied it some. This could never happen, however, because the program would never get to the `read`. We would be stuck in a situation called *deadlock.* You must be careful to avoid deadlock when using pipes, but don't be overly concerned: When you connect processes with single pipes used for one-way communication (like the shell does), deadlock (due to pipes) is *impossible.* Only the more exotic arrangements can cause deadlock.

Given that we have two processes, how can we connect them so that one can read from a pipe what the other writes? We can't. Once the processes are created they can't be connected, because there's no way for the process that creates the pipe to pass a file descriptor to the other process.[2] It can pass the file descriptor number, of course, but that number won't be valid in the other process. But if we make a pipe in one process *before* creating the other process, it will inherit the pipe file

2. Some systems have a nonportable way of doing this.

descriptors, and they'll be valid in both processes. Thus, two processes communicating over a pipe can be parent and child, or two children, or grandparent and grandchild, and so on. They must be related, however, and the pipe must be passed on at birth. In practice, this may be a severe limitation because, if a process dies, there's no way to recreate it and reconnect it to its pipes—the survivors must be killed too, and then the whole family has to be recreated.

In the following example one process (running the function `pipewrite`) makes a pipe, creates a child process (running `piperead`) that inherits it, and then writes some data to the pipe for the child to read. Although the child has inherited the necessary reading file descriptor, it still doesn't know its number, so the number is passed as an argument.

```
void pipewrite(void)
{
    int pfd[2];
    char fdstr[10];

    ec_neg1( pipe(pfd) )
    switch (fork()) {
    case -1:
        EC_FAIL
    case 0: /* child */
        ec_neg1( close(pfd[1]))
        snprintf(fdstr, sizeof(fdstr), "%d", pfd[0]);
        execlp("./piperead", "piperead", fdstr, (char *)NULL);
        EC_FAIL
    default: /* parent */
        ec_neg1( close(pfd[0]) )
        ec_neg1( write(pfd[1], "hello", 6) )
    }
    return;

EC_CLEANUP_BGN
    EC_FLUSH("pipewrite");
EC_CLEANUP_END
}
```

Here is the code for the child process:

```
int main(int argc, char *argv[])
{
    int fd;
    ssize_t nread;
    char s[100];
```

```
    fd = atoi(argv[1]);
    printf("reading file descriptor %d\n", fd);
    ec_neg1( nread = read(fd, s, sizeof(s)) )
    if (nread == 0)
        printf("EOF\n");
    else
        printf("read %ld bytes: %s\n", (long)nread, s);
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

And here is the output:

```
reading file descriptor 3
read 6 bytes: hello
```

Some comments about this example are needed:

- Since the child will only be reading the pipe, not writing it, it closes the writing end (`pfd[1]`) right away (the statement after `case 0` in the parent) to conserve file descriptors. (If the child were going to read more, it would be crucial to close the writing end, or the child would never get an end-of-file.)
- The parent has no use for the reading end of the pipe, so it closes `pdf[0]`.
- `snprintf` converts the reading file descriptor from an integer to a string so it can be used as a program argument.
- Since we know that the child program is in the current directory, we code its path as `./pread` to stop `execlp` from searching. This saves time, but more importantly, it prevents the accidental execution of some other `piperead` instead (no telling what might be in the user's path). We could have accomplished the same thing by using `execl`, which doesn't search.
- We didn't bother coding a `wait` for the child, since in this simple example the parent is going to exit right away.
- The child runs `piperead`, which just converts its argument back to an integer and reads from that file descriptor. (Remember, knowing that integer doesn't make the file descriptor valid—it's valid because it was inherited.)

In general, then, here is how to connect two processes with a pipe for one-way communication:

1. Create the pipe.
2. Fork to create the reading child.

3. In the child, close the writing end of the pipe and do any other preparations that are needed. (We'll see what these are in subsequent examples.)
4. In the child, execute the child program.
5. In the parent, close the reading end of the pipe.
6. If a second child is to write on the pipe, create it, make the necessary preparations, and execute its program. If the parent is to write, go ahead and write.

All our examples of one-way piping will follow this paradigm.

Now we can see why `fork` and `exec` are separate system calls. Why not a single system call to do both jobs, to save the overhead of `fork`? The two are separated to allow us to perform Step 3 above. We've already discovered some work to do between `fork` and `exec` (closing `pfd[1]`), and further on in this chapter we'll see more. We can't do this work before the `fork` because we don't want it to affect the parent (the parent must not close the writing end of the pipe). We don't want the child's program to do the work because we want programs to be oblivious to how they got invoked and how their inputs and outputs are connected (this is a cornerstone of UNIX philosophy). So we do it in the perfect place: in the child, executing code cloned from the parent. An added benefit is that the connection code is localized, making it easier to debug and modify.

To persist: Since there are only a few ways that processes are typically connected, why not have a single `fork-exec` system call with various options for interprocess connection? After all, there are other system calls with lots of options, so why not here? The answer is simply that the original designers of UNIX, Thompson and Ritchie, wanted to minimize the number of system calls. `fork` and `exec` are simple, and yet they allow the caller to arrange a wide variety of customized connections. So why kludge up another system call?[3]

The program `piperead` was specially designed to read from the file descriptor whose number was passed in. It's a strange program—no standard UNIX command works that way. Many programs do read a particular file descriptor, which they assume to be already open, but that file descriptor is fixed at 0 (the standard input, `STDIN_FILENO`). It doesn't have to be passed as an argument. Similarly, many programs are designed to write to file descriptor 1 (the standard output, `STDOUT_FILENO`). To connect commands as the shell does we somehow need to

3. But now we have it: `posix_spawn`, which we briefly mentioned in Section 5.5.

force `pipe` to return particular file descriptors in the `pfd` array: 0 for the reading end and 1 for the writing end. Alas, `pipe` offers no such feature. We might try closing 0 and 1 before calling `pipe`, to make them available, but this isn't safe because the standard says nothing about which end is which, and, furthermore, we can't usually afford to sacrifice our standard input and output just to make a pipe. So how do we perform the trick? We use `dup` or `dup2`.

6.3 dup and dup2 System Calls

dup—duplicate file descriptor

```
#include <unistd.h>

int dup(
    int fd              /* file descriptor to duplicate */
);
/* Returns new file descriptor or -1 on error (sets errno) */
```

dup2—duplicate file descriptor

```
#include <unistd.h>

int dup2(
    int fd,             /* file descriptor to duplicate */
    int fd2             /* file descriptor to use */
);
/* Returns new file descriptor or -1 on error (sets errno) */
```

`dup` duplicates an existing file descriptor, returning a new file descriptor that is open to the same file (or pipe, etc.). The two share the same file description (see Section 2.2), just as an inherited file descriptor shares the file description with the corresponding file descriptor in the parent. The call fails if the argument is bad (not open) or if no file descriptors are available.

`dup` takes the lowest-numbered available file descriptor, so if you know what's open, you can control what it returns. It's easier, though, to use `dup2`, which allows you to specify, with the `fd2` argument, what file descriptor to return. To make `fd2` available, `dup2` closes it if it has to.

`dup(fd)` is equivalent to

```
fcntl(fd, F_DUPFD, 0)
```

and `dup2(fd, fd2)` is equivalent to

```
close(fd2);
fcntl(fd, F_DUPFD, fd2);
```

except that `dup2` is atomic—if there's a problem with the duplication, `fd2` is not closed. We'll use `dup2`, rather than `dup`, in all our examples, because it's easier.

Since the file description is shared, there is only one benefit to having a second file descriptor: Its number is different and perhaps better suited to the caller's purposes. Suppose we create a pipe and then use `dup2` to make `STDIN_FILENO` (file descriptor 0) a duplicate of the reading end of the pipe (whose file descriptor might be 3, 4, 27, or whatever). If we then `exec` a program designed to read `STDIN_FILENO` (there are lots of them!), it will have been tricked into reading the pipe. A similar algorithm can be used to force `STDOUT_FILENO` to be the writing end of a pipe.

If the arguments to `dup2` are equal, it just returns that file descriptor without closing or duplicating anything. This is helpful in the case where, say, the reading end of the pipe is somehow already equal to `STDIN_FILENO`.

To illustrate `dup2`, here is an example that makes a pipe, creates a child to read it, arranges for the child's `STDIN_FILENO` to be the reading end of the pipe, and then invokes the `cat` command to read the pipe. Now that we're able to use `STDIN_FILENO`, we don't need a special program like `piperead` as we did in the example in the previous section.

```
void pipewrite2(void) /* has a bug */
{
    int pfd[2];
    pid_t pid;

    ec_neg1( pipe(pfd) )
    switch (pid = fork()) {
    case -1:
        EC_FAIL
    case 0: /* child */
        ec_neg1( dup2(pfd[0], STDIN_FILENO) )
        ec_neg1( close(pfd[0]))
        ec_neg1( close(pfd[1]))
        execlp("cat", "cat", (char *)NULL);
        EC_FAIL
    default: /* parent */
        ec_neg1( close(pfd[0]) )
        ec_neg1( write(pfd[1], "hello", 6) )
        ec_neg1( waitpid(pid, NULL, 0) )
    }
```

```
    return;

EC_CLEANUP_BGN
    EC_FLUSH("pipewrite2");
EC_CLEANUP_END
}
```

We got this unsurprising output:

```
hello
```

Note that in the child we closed both file descriptors we got from `pipe` after duplicating one of them. The reading end of the pipe is still open, but now its file descriptor is `STDIN_FILENO`. The parent closed `pfd[0]` (the reading end) because it didn't need it. The call to `waitpid` waits for `cat` to terminate; we had left it out of the `piperead` example in the previous section.

Unfortunately, after displaying "hello," the program hung, and we didn't get a shell prompt until we killed it with Ctrl-c. Do you see why? (Spoiler in next paragraph.)

The program hung because `cat`, which reads until it gets an end-of-file, didn't get one, so it didn't terminate. Recall that a 0 return from `read` occurs only when *all* writing file descriptors open to a pipe are closed, and at the time of the call to `waitpid`, `pfd[1]` was still open. Thus the parent and child were deadlocked. The solution is to close the writing end after writing the data:

```
    default: /* parent */
        ec_neg1( close(pfd[0]) )
        ec_neg1( write(pfd[1], "hello", 6) )
        ec_neg1( close(pfd[1]) )
        ec_neg1( waitpid(pid, NULL, 0) )
```

We don't have to limit ourselves to piping from parent to child. The next example implements the equivalent of the shell command line:

```
$ who | wc
```

to see how many users are logged in.

```
void who_wc(void)
{
    int pfd[2];
    pid_t pid1, pid2;

    ec_neg1( pipe(pfd) )
```

```
    switch (pid1 = fork()) {
    case -1:
        EC_FAIL
    case 0: /* first child */
        ec_neg1( dup2(pfd[1], STDOUT_FILENO) )
        ec_neg1( close(pfd[0]))
        ec_neg1( close(pfd[1]))
        execlp("who", "who", (char *)NULL);
        EC_FAIL
    }
    /* parent */
    switch (pid2 = fork()) {
    case -1:
        EC_FAIL
    case 0: /* second child */
        ec_neg1( dup2(pfd[0], STDIN_FILENO) )
        ec_neg1( close(pfd[0]))
        ec_neg1( close(pfd[1]))
        execlp("wc", "wc", "-l", (char *)NULL);
        EC_FAIL
    }
    /* still the parent */
    ec_neg1( close(pfd[0]) )
    ec_neg1( close(pfd[1]) )
    ec_neg1( waitpid(pid1, NULL, 0) )
    ec_neg1( waitpid(pid2, NULL, 0) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("who_wc");
EC_CLEANUP_END
}
```

This is the output:

```
1
```

Instead of `who` and `wc` both being children of the same parent, `wc` could be a child of `who`, which is just as easy to set up:

```
void who_wc2(void)
{
    int pfd[2];
    pid_t pid1, pid2;

    ec_neg1( pipe(pfd) )
    switch (pid1 = fork()) {
    case -1:
        EC_FAIL
```

```
    case 0: /* child */
        switch (pid2 = fork()) {
        case -1:
            EC_FAIL
        case 0: /* grandchild */
            ec_neg1( dup2(pfd[0], STDIN_FILENO) )
            ec_neg1( close(pfd[0]))
            ec_neg1( close(pfd[1]))
            execlp("wc", "wc", "-l", (char *)NULL);
            EC_FAIL
        }
        /* still the child */
        ec_neg1( dup2(pfd[1], STDOUT_FILENO) )
        ec_neg1( close(pfd[0]))
        ec_neg1( close(pfd[1]))
        execlp("who", "who", (char *)NULL);
        EC_FAIL
    }
    /* parent */
    ec_neg1( close(pfd[0]) )
    ec_neg1( close(pfd[1]) )
    ec_neg1( waitpid(pid1, NULL, 0) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("who_wc2");
EC_CLEANUP_END
}
```

Two points about this example:

- The child has to create the grandchild before closing the reading file descriptor so the grandchild will inherit it.
- The parent waits only for `who` (`pid1`) to terminate. It can't wait for `wc` because `wc` is not its child. The child running `who` can't wait for `wc` either because the `who` program hasn't been designed to do that. Putting a call to `waitpid` after the call that `exec`s `wc` won't help because on a successful `exec` all the code is overwritten. What happens (as explained in Section 5.8) is that when `who` (`wc`'s parent) terminates, a system process will inherit `wc` and will wait for it.

We could just as easily have reversed things and made `wc` the parent of `who` (see Exercise 6.11).

Of course, it's much easier to use a shell to set up a command line like this than it is to write a custom program. In the next section we'll code such a shell.

6.4 A Real Shell

Our shell is a subset of the typical UNIX shell that you probably use. It has these features:

- A *simple command* consists of a command name followed by an optional sequence of arguments separated by spaces or tabs, each of which is a single word or a string surrounded by double quotes (`"`). If quoted, an argument may include any otherwise special characters (`|`, `;`, `&`, `>`, `<`, space, tab, and newline). An included quote or backslash must be preceded with a backslash (`\"` or `\\`). A command may have up to 50 arguments, each of which may have up to 500 characters.
- A simple command's standard input may be redirected to come from a file by preceding the file name with `<`. Similarly, the standard output may be redirected with `>`, which truncates the output file. If the output redirection symbol is `>>`, the output is appended to the file. An output file is created if it doesn't exist.
- A *pipeline* consists of a sequence of one or more simple commands separated with bars (`|`). Each except the last simple command in a pipeline has its standard output connected, via a pipe, to the standard input of its right neighbor.
- A pipeline is terminated with a newline, a semicolon (`;`), or an ampersand (`&`). In the first two cases the shell waits for the rightmost simple command to terminate before continuing. In the ampersand case, the shell does not wait. It reports the process number of each simple command in the pipeline, and each simple command is run with interrupt and quit signals ignored.
- Built-in commands are assignment, `set`, and `cd`. The first two of these were described in Section 5.4. `cd` works in the familiar way.

The first step is to parse input lines into *tokens,* which are groups of characters that form syntactic units; examples are words, quoted strings, and special symbols such as `&` and `>>`. Each token is represented by a symbolic constant, as follows:

`T_WORD` An argument or file name. If quoted, the quotes are removed after the token is recognized.

`T_BAR` The symbol `|`.

`T_AMP` The symbol `&`.

`T_SEMI` The symbol `;`.

`T_GT` The symbol `>`.

`T_GTGT` The symbol `>>`.

`T_LT` The symbol `<`.

`T_NL` A newline.

`T_EOF` A special token signifying that the end-of-file has been reached. If the standard input is a terminal, the user has typed an EOT (Ctrl-d).

`T_ERROR` A special token signifying an error.

The job of a *lexical analyzer* is to read the input and assemble the characters into tokens. Each time it is called, one token is returned. If the token is `T_WORD`, a string containing the actual characters composing it is also returned. (For the other tokens the actual characters are obvious.) The lexical analyzer should bypass irrelevant characters, such as spaces separating arguments, without returning anything.[4]

Our lexical analyzer is a finite-state-machine: As characters are read they are either recognized immediately as tokens or they are accumulated (characters of a word, for example). With each character, the lexical analyzer can switch into a new state which serves to remember what it is doing and how characters are to be interpreted. For example, when accumulating a quoted string, spaces are treated differently than when they appear outside quotes. For our shell we need four states:

`NEUTRAL` The starting state for each call to the lexical analyzer. Spaces and tabs are skipped. The characters `|`, `&`, `;`, `<`, and newline are recognized immediately as tokens. The character `>` causes a switch to state `GTGT`, which will see if it is followed by another `>`, since `>` and `>>` are two different tokens. A quote causes a switch to state `INQUOTE`, which gathers a quoted string. Anything else is taken as the beginning of an unquoted word; the character is saved in a buffer and the state is switched to `INWORD`.

4. Most UNIX systems include a command called `lex` that can automatically generate a lexical analyzer from a description of the tokens it is to recognize. Because `lex` is complicated to use, and because the lexical analyzers it generates are sometimes big and slow, it's usually preferable to code lexical analyzers by hand. They aren't difficult programs to write.

`GTGT` This state means that `>` was just read. If the next character is also `>`, the token `T_GTGT` is returned. Otherwise, `T_GT` will be returned. But first, we've read one character too many, so we put it back on the input with the Standard C function `ungetc`.

`INQUOTE` This state means that a starting quote was read. We accumulate characters into a buffer until the closing quote is read. Then we return the token `T_WORD` and the accumulated string. Special steps must be taken to process the escape character `\`.

`INWORD` This state means that the first character of a word was read and has been put into the buffer. We keep accumulating characters until a nonword character is read (`|`, say). We put the gratuitous character back on the input and return the token `T_WORD`.

With this explanation, the code for our lexical analyzer, `gettoken`, should be readily understandable:

```
typedef enum {T_WORD, T_BAR, T_AMP, T_SEMI, T_GT, T_GTGT, T_LT,
  T_NL, T_EOF, T_ERROR} TOKEN;

static TOKEN gettoken(char *word, size_t maxword)
{
    enum {NEUTRAL, GTGT, INQUOTE, INWORD} state = NEUTRAL;
    int c;
    size_t wordn = 0;

    while ((c = getchar()) != EOF) {
        switch (state) {
        case NEUTRAL:
            switch (c) {
            case ';':
                return T_SEMI;
            case '&':
                return T_AMP;
            case '|':
                return T_BAR;
            case '<':
                return T_LT;
            case '\n':
                return T_NL;
            case ' ':
            case '\t':
                continue;
            case '>':
                state = GTGT;
                continue;
```

```
            case '"':
                state = INQUOTE;
                continue;
            default:
                state = INWORD;
                ec_false( store_char(word, maxword, c, &wordn) )
                continue;
            }
        case GTGT:
            if (c == '>')
                return T_GTGT;
            ungetc(c, stdin);
            return T_GT;
        case INQUOTE:
            switch (c) {
            case '\\':
                if ((c = getchar()) == EOF)
                    c = '\\';
                ec_false( store_char(word, maxword, c, &wordn) );
                continue;
            case '"':
                ec_false( store_char(word, maxword, '\0', &wordn) )
                return T_WORD;
            default:
                ec_false( store_char(word, maxword, c, &wordn) )
                continue;
            }
        case INWORD:
            switch (c) {
            case ';':
            case '&':
            case '|':
            case '<':
            case '>':
            case '\n':
            case ' ':
            case '\t':
                ungetc(c, stdin);
                ec_false( store_char(word, maxword, '\0', &wordn) )
                return T_WORD;
            default:
                ec_false( store_char(word, maxword, c, &wordn) )
                continue;
            }
        }
    }
    ec_false( !ferror(stdin) )
    return T_EOF;
```

```
EC_CLEANUP_BGN
    return T_ERROR;
EC_CLEANUP_END
}

static bool store_char(char *word, size_t maxword, int c, size_t *np)
{
    errno = E2BIG;
    ec_false( *np < maxword )
    word[(*np)++] = c;
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

When coding a large program, it's convenient to debug it in pieces. This not only provides early feedback, but it makes finding bugs easier since there are fewer lines of code to search. Here's a test program for `gettoken`:

```
int main(void)
{
    char word[200];

    while (1)
        switch (gettoken(word, sizeof(word))) {
        case T_WORD:
            printf("T_WORD <%s>\n", word);
            break;
        case T_BAR:
            printf("T_BAR\n");
            break;
        case T_AMP:
            printf("T_AMP\n");
            break;
        case T_SEMI:
            printf("T_SEMI\n");
            break;
        case T_GT:
            printf("T_GT\n");
            break;
        case T_GTGT:
            printf("T_GTGT\n");
            break;
        case T_LT:
            printf("T_LT\n");
            break;
```

```
        case T_NL:
            printf("T_NL\n");
            break;
        case T_EOF:
            printf("T_EOF\n");
            exit(EXIT_SUCCESS);
        case T_ERROR:
            printf("T_ERROR\n");
            exit(EXIT_SUCCESS);
        }
}
```

When we ran this program and typed this line (followed by an EOT)

```
sort <inf | pr -h "Sept. Results" >>outf&
```

we got this output:

```
T_WORD <sort>
T_LT
T_WORD <inf>
T_BAR
T_WORD <pr>
T_WORD <-h>
T_WORD <Sept. Results>
T_GTGT
T_WORD <outf>
T_AMP
T_NL
T_EOF
```

This test doesn't prove that `gettoken` is correct, but it's encouraging enough for us to go on with more of the shell.

The next step is to write a function to process a simple command, which is terminated by `|`, `&`, `;`, or a newline. We'll call this function `command`. Arguments are simply entered into an `argv` array for later use in a call to `execvp`. There are three possibilities for the standard input: the default (`STDIN_FILENO`), a file (if `<` was present), or the reading end of a pipe (if this simple command was *preceded* by `|`). The standard output has four possibilities: the default (`STDOUT_FILENO`), a file to be created or truncated (if `>` was present), a file to be created or appended to (if `>>` was present), or the writing end of a pipe (if this simple command was *followed* by a `|`). The tokens received from `gettoken` tell us which possibilities obtain.

As `command` processes the tokens, it uses these variables to record the standard input and output situation for a simple command:

`srcfd` The source file descriptor, initially `STDIN_FILENO`. If input is redirected with <, `srcfd` is set to –1 and `srcfile` records the file name. If the input is a pipe, `srcfd` will be set to a file-descriptor number other than `STDIN_FILENO`.

`srcfile` The source file, used only when the simple command includes a <.

`dstfd` The destination file descriptor, initially `STDOUT_FILENO`. If output is redirected with > or >>, it is set to –1 and `dstfile` records the file name. Like `srcfd`, it's set to a file descriptor other than `STDOUT_FILENO` if the output is a pipe.

`dstfile` The output file, used only when the simple command includes a > or >>.

`append` This Boolean variable is set to `true` only if output was redirected with >>.

`makepipe` This is an argument to `command`. If `true`, the caller is requesting that `command` make a pipe, use the reading end as its standard input, and pass the file descriptor for the writing end back to the caller, who will use it for its standard output. This scheme will be explained shortly.

As it goes through the tokens, `command` also checks for logical errors: two occurrences of < or >, the occurrence of > when the simple command terminates with |, the occurrence of < when the simple command is preceded with |, and so on. It's interesting that the typical UNIX shell doesn't check for some of these anomalies: You can actually run a command line like this:

```
$ who >outf | wc
```

The function `command` returns the token that terminated the pipeline because the type of terminator affects later processing: If the terminator is `&`, the rightmost simple command is not waited for, and all simple commands in the pipeline are run with interrupt and quit signals ignored. If the rightmost simple command is to be waited for, its process-ID must be returned, too, through the argument `wpid`. If the terminator is a newline, the user is prompted for a new command.

The most subtle thing about `command` is that it calls itself recursively when a simple command is followed by |. Recursive calls continue until one of the other terminators (`;`, `&`, or newline) is reached. Each recursive call is responsible for actually making the pipe (with the `pipe` system call), and its argument `makepipe`

is therefore set to `true`. The writing pipe file descriptor is passed back (through another argument, `pipefdp`). A simple command is not actually invoked (with the function `invoke`) until after the recursive call returns, since it can't be invoked until the writing end of the pipe is available. Also, as stated above, the pipeline terminator must be known before any of the constituent simple commands can be invoked so that `invoke` will know what to do about signals. Thus pipelines are processed from left to right, but the simple commands are invoked from right to left. There is one call to `command`, with `makepipe` set to `false`, for the leftmost simple command; and then one more recursive call, with `makepipe` set to `true`, for each additional simple command. As the stack of command calls returns, the simple commands are invoked—all needed information is available then.

Here is the code for `command`. It's worth careful study.

```
#define MAXARG 50       /* max args in command */
#define MAXFNAME 500    /* max chars in file name */
#define MAXWORD 500     /* max chars in arg */

static TOKEN command(pid_t *wpid, bool makepipe, int *pipefdp)
{
    TOKEN token, term;
    int argc, srcfd, dstfd, pid, pfd[2] = {-1, -1};
    char *argv[MAXARG], srcfile[MAXFNAME] = "", dstfile[MAXFNAME] = "";
    char word[MAXWORD];
    bool append;

    argc = 0;
    srcfd = STDIN_FILENO;
    dstfd = STDOUT_FILENO;
    while (true) {
        switch (token = gettoken(word, sizeof(word))) {
        case T_WORD:
            if (argc >= MAXARG - 1) {
                fprintf(stderr, "Too many args\n");
                continue;
            }
            if ((argv[argc] = malloc(strlen(word) + 1)) == NULL) {
                fprintf(stderr, "Out of arg memory\n");
                continue;
            }
            strcpy(argv[argc], word);
            argc++;
            continue;
```

```
case T_LT:
    if (makepipe) {
        fprintf(stderr, "Extra <\n");
        break;
    }
    if (gettoken(srcfile, sizeof(srcfile)) != T_WORD) {
        fprintf(stderr, "Illegal <\n");
        break;
    }
    srcfd = -1;
    continue;
case T_GT:
case T_GTGT:
    if (dstfd != STDOUT_FILENO) {
        fprintf(stderr, "Extra > or >>\n");
        break;
    }
    if (gettoken(dstfile, sizeof(dstfile)) != T_WORD) {
        fprintf(stderr, "Illegal > or >>\n");
        break;
    }
    dstfd = -1;
    append = token == T_GTGT;
    continue;
case T_BAR:
case T_AMP:
case T_SEMI:
case T_NL:
    argv[argc] = NULL;
    if (token == T_BAR) {
        if (dstfd != STDOUT_FILENO) {
            fprintf(stderr, "> or >> conflicts with |\n");
            break;
        }
        term = command(wpid, true, &dstfd);
        if (term == T_ERROR)
            return T_ERROR;
    }
    else
        term = token;
    if (makepipe) {
        ec_neg1( pipe(pfd) )
        *pipefdp = pfd[1];
        srcfd = pfd[0];
    }
    ec_neg1( pid = invoke(argc, argv, srcfd, srcfile, dstfd,
      dstfile, append, term == T_AMP, pfd[1]) )
    if (token != T_BAR)
        *wpid = pid;
```

```
            if (argc == 0 && (token != T_NL || srcfd > 1))
                fprintf(stderr, "Missing command\n");
            while (--argc >= 0)
                free(argv[argc]);
            return term;
        case T_EOF:
            exit(EXIT_SUCCESS);
        case T_ERROR:
            return T_ERROR;
        }
    }

EC_CLEANUP_BGN
    return T_ERROR;
EC_CLEANUP_END
}
```

Once `command` has sensed the need for a pipe by encountering the token `T_BAR`, why does it ask the *next* call to `command` to make the pipe instead of making it itself and just passing on the reading file descriptor? It is to conserve file descriptors. If the `pipe` system call were called before the recursive call to `command`, instead of after, each level of recursion—each simple command—would tie up two pipe file descriptors that could not be closed until after the recursive call returned. Since on some systems file descriptors are in short supply, pipelines would be limited to somewhat less than half that number of simple commands. By asking the reader to make the pipe, we can call `pipe` just before we call `invoke` (which we'll get to shortly), if our caller asked us to make one, thereby allowing pipelines to be of any length.

The child that `invoke` will create has no need for the writing end of the pipe (`pfd[1]`), so it's passed as the last argument to `invoke` so it can be closed in the child; we'll see this a bit later.

Here's the `main` program that makes the first call to `command`. It's modeled on the `main` program for the one-shot shell in Section 5.4.

```
int main(void)
{
    pid_t pid;
    TOKEN term = T_NL;

    ignore_sig();
    while (true) {
        if (term == T_NL)
            printf("%s", PROMPT);
        term = command(&pid, false, NULL);
```

```
        if (term == T_ERROR) {
            fprintf(stderr, "Bad command\n");
            EC_FLUSH("main--bad command")
            term = T_NL;
        }
        if (term != T_AMP && pid > 0)
            wait_and_display(pid);
        fd_check();
    }
}
```

The call to `ignore_sig` causes interrupt and quit signals to be ignored—we don't want to kill our shell when we hit the interrupt or quit keys. We'll show the code for `ignore_sig` in Section 9.1.6. The pipeline terminator returned by `command` tells us whether to wait for the rightmost simple command to terminate (we'll see this version of `wait_and_display` shortly) and whether to prompt.

We want to make sure that with all the redirection and piping we don't miss closing any file descriptors, so we call `fd_check` each time we process a command to make sure that only the standard file descriptors are open. (If we wanted to, we could remove `fd_check` from a production version of this shell.) We check only the first 20 file descriptors, as that's enough to reveal any failure-to-close bugs:

```
static void fd_check(void)
{
    int fd;
    bool ok = true;

    for (fd = 3; fd < 20; fd++)
        if (fcntl(fd, F_GETFL) != -1 || errno != EBADF) {
            ok = false;
            fprintf(stderr, "*** fd %d is open ***\n", fd);
        }
    if (!ok)
        _exit(EXIT_FAILURE);
}
```

`command` calls `invoke` to invoke a simple command. It passes on the command arguments (`argc` and `argv`) and the source and destination variables described earlier (`srcfd`, `srcfile`, `dstfd`, `dstfile`, and `append`). The next-to-last argument tells invoke whether the simple command is to be run in the background; the last argument is the file descriptor to close in the child, as we already described. The `fork` and `exec` scheme used by `invoke` should by now be familiar:

```
static pid_t invoke(int argc, char *argv[], int srcfd, const char *srcfile,
  int dstfd, const char *dstfile, bool append, bool bckgrnd, int closefd)
{
    pid_t pid;
    char *cmdname, *cmdpath;

    if (argc == 0 || builtin(argc, argv, srcfd, dstfd))
        return 0;
    switch (pid = fork()) {
    case -1:
        fprintf(stderr, "Can't create new process\n");
        return 0;
    case 0:
        if (closefd != -1)
            ec_neg1( close(closefd) )
        if (!bckgrnd)
            ec_false( entry_sig() )
        redirect(srcfd, srcfile, dstfd, dstfile, append, bckgrnd);
        cmdname = strchr(argv[0], '/');
        if (cmdname == NULL)
            cmdname = argv[0];
        else
            cmdname++;
        cmdpath = argv[0];
        argv[0] = cmdname;
        execvp(cmdpath, argv);
        fprintf(stderr, "Can't execute %s\n", cmdpath);
        _exit(EXIT_FAILURE);
    }
    /* parent */
    if (srcfd > STDOUT_FILENO)
        ec_neg1( close(srcfd) )
    if (dstfd > STDOUT_FILENO)
        ec_neg1( close(dstfd) )
    if (bckgrnd)
        printf("%ld\n", (long)pid);
    return pid;

EC_CLEANUP_BGN
    if (pid == 0)
        _exit(EXIT_FAILURE);
    return -1;
EC_CLEANUP_END
}
```

The horsing around with `cmdname` and `cmdpath` is caused by the need to keep the whole path that might have been typed as the first argument to `execvp`, but to make `argv[0]` point to only the file-name part.

The command to be invoked may be built in. If so, `builtin` (to be shown shortly) returns `true`. If not, a child process is created to run the simple command. Recall that interrupt and quit signals are already ignored. If the command is not to be run in the background, they are restored to the way they were on entry to the shell with a call to `entry_sig` (detailed in Section 9.1.6).

`invoke` calls `redirect` to redirect I/O and to ensure that the source and destination are duped to be `STDIN_FILENO` and `STDOUT_FILENO`, if necessary. Here is the code for `redirect`:

```
static void redirect(int srcfd, const char *srcfile, int dstfd,
  const char *dstfile, bool append, bool bckgrnd)
{
    int flags;

    if (srcfd == STDIN_FILENO && bckgrnd) {
        srcfile = "/dev/null";
        srcfd = -1;
    }
    if (srcfile[0] != '\0')
        ec_neg1( srcfd = open(srcfile, O_RDONLY, 0) )
    ec_neg1( dup2(srcfd, STDIN_FILENO) )
    if (srcfd != STDIN_FILENO)
        ec_neg1( close(srcfd) )
    if (dstfile[0] != '\0') {
        flags = O_WRONLY | O_CREAT;
        if (append)
            flags |= O_APPEND;
        else
            flags |= O_TRUNC;
        ec_neg1( dstfd = open(dstfile, flags, PERM_FILE) )
    }
    ec_neg1( dup2(dstfd, STDOUT_FILENO) )
    if (dstfd != STDOUT_FILENO)
        ec_neg1( close(dstfd) )
    fd_check();
    return;

EC_CLEANUP_BGN
    _exit(EXIT_FAILURE); /* we are in child */
EC_CLEANUP_END
}
```

If a background command does not have its standard input redirected to come from a file or a pipe, we take the precaution of redirecting it to the special file /dev/null, which gives an immediate end-of-file when read. The rest of `redirect` should be clear.

`wait_and_display` (called from `main`) is an extension to code presented in Section 5.8. It's told what process to wait for, but during the wait it may learn of other terminations. If so, it uses `display_status` to print their process IDs and a description of the reason for their termination. When the designated process terminates, `wait_and_display` returns after displaying its status. Here's the code:

```
static bool wait_and_display(pid_t pid)
{
    pid_t wpid;
    int status;

    do {
        ec_neg1( wpid = waitpid(-1, &status, 0) )
        display_status(wpid, status);
    } while (wpid != pid);
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Finally, here is `builtin`. Most of it is taken from Section 5.4; we've added code to handle the `cd` command. `cd` without arguments changes to the directory whose path is given by the `HOME` environment variable.

```
static bool builtin(int argc, char *argv[], int srcfd, int dstfd)
{
    char *path;

    if (strchr(argv[0], '=') != NULL)
        asg(argc, argv);
    else if (strcmp(argv[0], "set") == 0)
        set(argc, argv);
    else if (strcmp(argv[0], "cd") == 0) {
        if (argc > 1)
            path = argv[1];
        else if ((path = getenv("HOME")) == NULL)
            path = ".";
        if (chdir(path) == -1)
            fprintf(stderr, "%s: bad directory\n", path);
    }
    else
        return false;
    if (srcfd != STDIN_FILENO || dstfd != STDOUT_FILENO)
        fprintf(stderr, "Illegal redirection or pipeline\n");
    return true;
}
```

You may have observed that some errors discovered by our shell are handled by the heavy "ec" machinery, while others just cause a message to be printed, after which the shell keeps running. We've treated the "impossible" errors as serious ones, since if they occur it means that the operating system is mortally wounded. This is a compromise: We certainly don't want to ignore these errors, because the impossible has been known to happen, but we don't want to code to recover from a situation that will probably never occur. Sometimes we guess wrong—a serious error keeps occurring because it's not impossible after all. Then we have to revise the code to handle the error differently.

I won't show any example of this shell in use. I don't have to because it behaves just like those of the shell you use every day.

6.5 Two-Way Communication with Unidirectional Pipes

Now we want to move beyond one-way pipeline communication, as used by the shell, to two-way. The typical shell offers no notation to set up two-way communication between processes. Two-way pipelines are set up from C programs.

We'll start with a fairly simple example. From within a program we want to invoke the `sort` command to sort some data. Of course, we could do it like this:

```
system("sort <datafile >outfile");
```

Then we could read the contents of outfile to access the sorted data. We don't want to do it that way, however—we want to pipe the data to `sort` and have `sort` pipe the sorted output back to us. Since `sort` can read and write its standard input and output (it's a filter), we should be able to use it the way we want. We already know how to force an arbitrary file descriptor to be the standard input or output of a process.

Just as a baby learns to fall down before learning to walk, we'll begin by doing the job incorrectly. That way we'll learn more about how to do it properly than if we just presented the solution right away.

Since every pipe has both a reading end and a writing end, and since both file descriptors are inherited by a child process, we'll use just one pipe. `sort` will

read it to get its input and write it to send back the sorted output. The parent has file descriptors that access the pipe too. It will write the unsorted data to the pipe and read the sorted data from the pipe. Here's a program that reads data from the file datafile and invokes sort to sort it; the sorted data is printed:

```
void fsort0(void)  /* wrong */
{
    int pfd[2], fd;
    ssize_t nread;
    pid_t pid;
    char buf[512];

    ec_neg1( pipe(pfd) )
    ec_neg1( pid = fork() )
    if (pid == 0) { /* child */
        ec_neg1( dup2(pfd[0], STDIN_FILENO) )
        ec_neg1( close(pfd[0]) )
        ec_neg1( dup2(pfd[1], STDOUT_FILENO) )
        ec_neg1( close(pfd[1]) )
        execlp("sort", "sort", (char *)NULL);
        EC_FAIL
    }
    /* parent */
    ec_neg1( fd = open("datafile", O_RDONLY) )
    while (true) {
        ec_neg1( nread = read(fd, buf, sizeof(buf)) )
        if (nread == 0)
            break;
        ec_neg1( write(pfd[1], buf, nread) )
    }
    ec_neg1( close(fd) )
    ec_neg1( close(pfd[1]) )
    while (true) {
        ec_neg1( nread = read(pfd[0], buf, sizeof(buf)) )
        if (nread == 0)
            break;
        ec_neg1( write(STDOUT_FILENO, buf, nread) )
    }
    ec_neg1( close(pfd[0]) )
    ec_neg1( waitpid(pid, NULL, 0) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("fsort0");
EC_CLEANUP_END
}
```

This is what's in datafile:

```
peach
apple
orange
strawberry
plum
pear
cherry
banana
apricot
tomato
pineapple
mango
```

When I ran this program, it printed the fruits just as they appeared in datafile—unsorted—and then hung. I had to hit the interrupt key to kill it. What went wrong?

There were two problems with using just one pipe. First of all, after writing the unsorted data to the pipe, the parent immediately began to read the pipe, assuming that it would read the output of `sort`. It got there before `sort` did, however, and just read its own output right back! So the data was printed in its unsorted form. This is reminiscent of the first example in Section 6.2.3.

The second problem caused the deadlock. The child process, running `sort`, began to read its standard input, which happened to be empty since its parent had already emptied it. Empty or not, `sort` would have blocked in a `read` system call waiting for an end-of-file, which would occur only when the writing end was closed. Sure enough, the parent had already closed the writing end, but the *child* still had it open—after all, the child was supposed to write its output there. So the child was stuck. Generally, any filter tricked into reading and writing the same pipe will deadlock.

One might try to fix the first problem by having the parent wait for the child to terminate before reading the pipe. At first, this sounds appealing. It won't work, however, if the child's output fills the pipe, which it might if we're sorting a large amount of data, because the child will block in the `write` system call. With the parent blocked in `waitpid`, we'll have deadlock again.

Next, one might try some semaphores to synchronize things without causing deadlock, but this is overkill. The problem goes away completely if one just uses *two* pipes, each of which handles only one-way traffic. One pipe handles data

flowing into `sort`, and the other pipe handles data flowing back. Here is a rewrite of `fsort0` that works correctly:

```
void fsort(void)
{
    int pfdout[2], pfdin[2], fd;
    ssize_t nread;
    pid_t pid;
    char buf[512];

    ec_neg1( pipe(pfdout) )
    ec_neg1( pipe(pfdin) )
    ec_neg1( pid = fork() )
    if (pid == 0) { /* child */
        ec_neg1( dup2(pfdout[0], STDIN_FILENO) )
        ec_neg1( close(pfdout[0]) )
        ec_neg1( close(pfdout[1]) )
        ec_neg1( dup2(pfdin[1], STDOUT_FILENO) )
        ec_neg1( close(pfdin[0]) )
        ec_neg1( close(pfdin[1]) )
        execlp("sort", "sort", (char *)NULL);
        EC_FAIL
    }
    /* parent */
    ec_neg1( close(pfdout[0]) )
    ec_neg1( close(pfdin[1]) )
    ec_neg1( fd = open("datafile", O_RDONLY) )
    while (true) {
        ec_neg1( nread = read(fd, buf, sizeof(buf)) )
        if (nread == 0)
            break;
        ec_neg1( write(pfdout[1], buf, nread) )
    }
    ec_neg1( close(fd) )
    ec_neg1( close(pfdout[1]) )
    while (true) {
        ec_neg1( nread = read(pfdin[0], buf, sizeof(buf)) )
        if (nread == 0)
            break;
        ec_neg1( write(STDOUT_FILENO, buf, nread) )
    }
    ec_neg1( close(pfdin[0]) )
    ec_neg1( waitpid(pid, NULL, 0) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("fsort");
EC_CLEANUP_END
}
```

Now we get:

```
apple
apricot
banana
cherry
mango
orange
peach
pear
pineapple
plum
strawberry
tomato
```

Not only is the list sorted, but the program stopped all by itself! The interesting thing about the correct version is that although it uses one more pipe, no additional file descriptors are consumed. In both versions parent and child need to read and write to each other, so each needs two pipe file descriptors. In the second version the parent can close the reading end of `pfdout` and the writing end of `pfdin`.

In general, deadlock is still possible with two pipes, although not in our example using `sort`. Deadlock would occur if the parent blocks writing its output pipe because it is full, and the child, instead of emptying it, writes enough output back to the parent to block on the other pipe. Each case must be examined carefully to ensure that it is always deadlock free, and not only for small amounts of test data.

There are other ways to get deadlocked. To explore one of these, we'll look at a more complex example of two-way interprocess communication. Here the child is the standard line editor, `ed`. The parent is using the editor as a server sending editor command lines to it and getting the output back. Such an arrangement might be used by a visual editor, where the parent handles the keyboard and screen but lets `ed` do the actual editing. I don't have space to show a visual editor, so my example will have to be ridiculously simple. It's an interactive search program much like `grep`. Here's a sample session (what I typed is underlined):

```
$ search

File? datafile

Search pattern? ^a
apple
apricot
```

```
Search pattern? apple
apple
pineapple

Search pattern? o$
tomato
mango

Search pattern? EOT
$
```

The fruit data file is from the sorting examples, above.

When I used `sort`, I knew when to stop reading its output: when I reached an end-of-file. The situation was simple because `sort` reads all its input, writes all its output, and then terminates. The editor, however, is interactive. It reads some input, may or may not write some output of indeterminate length, and then goes back to read some more input. We can capture its output by making its standard output a pipe, but how do we know how much to read at a time? We can't wait for an end-of-file, for the editor won't close its output file descriptor until it terminates. If we read too far we'll deadlock, and if we don't read far enough we'll lose synchronization between a command and its results.

If we could change the editor, we would have it send an unambiguous line of data whenever it's ready for more input. In fact, the editor does have the ability to prompt the user for input (enabled by the `P` command), but the prompt character is `*`, which is hardly unambiguous. (Although you can change the prompt; see Exercise 6.12.) Somehow we need to get the editor to tell us when it's done outputting the results of each command.

We'll use a trick, a kludge if there ever was one: After each command, issue an `r` (read file) command with a nonexistent file name and look for the resulting error message from `ed`. When this message shows up, we know the editor has responded to the bad `r` command and is ready for another command. The nonexistent file name should be such that the error message is unambiguous. It's probably best to use a name containing control characters, which rarely appear in text files, but for clarity we'll use the name "end-of-file." To see the exact form of the message, we run the editor:

```
$ ed
r end-of-file
?end-of-file
q
$
```

So we have to look for "?end-of-file."

To make things easier, we'll code functions to handle interaction with the editor. At the start of processing we call `edinvoke` to invoke the editor and set up the pipes. Instead of using the file descriptors directly, which would require us to do I/O with `read` and `write`, we create Standard C `FILE` pointers instead, using `fdopen`. We can then write to the editor on `sndfp` and read from the editor on `rcvfp`. Here is `edinvoke`, which is a lot like the first part of `fsort`:

```
static FILE *sndfp, *rcvfp;

static bool edinvoke(void)
{
    int pfdout[2], pfdin[2];
    pid_t pid;

    ec_neg1( pipe(pfdout) )
    ec_neg1( pipe(pfdin) )
    switch (pid = fork()) {
    case -1:
        EC_FAIL
    case 0:
        ec_neg1( dup2(pfdout[0], STDIN_FILENO) )
        ec_neg1( dup2(pfdin[1], STDOUT_FILENO) )
        ec_neg1( close(pfdout[0]) )
        ec_neg1( close(pfdout[1]) )
        ec_neg1( close(pfdin[0]) )
        ec_neg1( close(pfdin[1]) )
        execlp("ed", "ed", "-", (char *)NULL);
        EC_FAIL
    }
    ec_neg1( close(pfdout[0]) )
    ec_neg1( close(pfdin[1]) )
    ec_null( sndfp = fdopen(pfdout[1], "w") )
    ec_null( rcvfp = fdopen(pfdin[0], "r") )
    return true;

EC_CLEANUP_BGN
    if (pid == 0) {
        EC_FLUSH("edinvoke");
        _exit(EXIT_FAILURE);
    }
    return false;
EC_CLEANUP_END
}
```

Note that we called `_exit` on an error in the child, rather than `exit`, but then had to call `EC_FLUSH` to get the error message displayed, exactly as we did in Section 5.6.

To write and read the pipe we use `edsnd`, `edrcv`, and `turnaround`. `edrcv` returns `false` when no more editor output is available, which fact it knows because it has found the forced error message. (It treats an actual end-of-file, indicated by a `NULL` return from `fgets`, as an error.) To make sure the error message is there, `turnaround` must be called to switch from sending to receiving. Here are these three functions:

```
static bool edsnd(const char *s)
{
    ec_eof( fputs(s, sndfp) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

static bool edrcv(char *s, size_t smax)
{
    ec_null( fgets(s, smax, rcvfp) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}

static bool turnaround(void)
{
    ec_false( edsnd("r end-of-file\n") )
    ec_eof( fflush(sndfp) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

We flush `sndfp` so that the editor gets all our output because normally output to pipes is buffered.

Since we'll usually want to display everything the editor has to say, a function to do that will come in handy:

```
static bool rcvall(void)
{
    char s[200];

    ec_false( turnaround() )
    while (true) {
        ec_false( edrcv(s, sizeof(s)) )
        if (strcmp(s, "?end-of-file\n") == 0)
            break;
        printf("%s", s);
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Last, the `main` program that conducts the dialogue with the user and communicates with the editor:

```
int main(void)
{
    char s[100], line[200];
    bool eof;

    ec_false( prompt("File", s, sizeof(s), &eof) )
    if (eof)
        exit(EXIT_SUCCESS);
    ec_false( edinvoke() )
    snprintf(line, sizeof(line), "e %s\n", s);
    ec_false( edsnd(line) )
    ec_false( rcvall() );
    while (true) {
        ec_false( prompt("Search pattern", s, sizeof(s), &eof) )
        if (eof)
            break;
        snprintf(line, sizeof(line), "g/%s/p\n", s);
        ec_false( edsnd(line) )
        ec_false( rcvall() );
    }
    ec_false( edsnd("q\n") )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

```
static bool prompt(const char *msg, char *result, size_t resultmax,
  bool *eofp)
{
    char *p;

    printf("\n%s? ", msg);
    if (fgets(result, resultmax, stdin) == NULL) {
        if (ferror(stdin))
            EC_FAIL
        *eofp = true;
    }
    else {
        if ((p = strrchr(result, '\n')) != NULL)
            *p = '\0';
        *eofp = false;
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Is this program deadlock free? Since we know that the capacity of a pipe is at least 512 bytes, the parent will never block while writing to the editor's input pipe because our editor commands are quite short. The editor will never block, except temporarily, writing back to the parent because the parent will read everything it has to say. There's no reason why the bad-file message should be missed, unless somebody actually makes a file with that name, which they had better not. Not much of a proof, but at least the obvious pitfalls have been avoided.

The example output shown was generated on Solaris. The program didn't work at all on FreeBSD, Darwin, or Linux because their version of `ed` is different. See Exercise 6.12 if you're curious about the details.

6.6 Two-Way Communication with Bidirectional Pipes

I said when I first presented pipes (Section 6.2.1) that the first file descriptor (`pfd[0]`) is opened for reading, the second (`pfd[1]`) is opened for writing, and whatever is written on `pfd[1]` can be read from `pfd[0]`. Then I showed several useful examples where one process is the writer and another the reader, including some using two pipes (each unidirectional) for two-way communication, much like a divided highway made up, in effect, of two one-way roads.

Everything I said about pipes is true, but that doesn't mean that the file descriptors are opened *only* for reading or *only* for writing. To see what the story is, here's a little program that shows the access mode for a pipe's file descriptors:

```
void pipe_access_mode(void)
{
    int pfd[2], flags, i;

    ec_neg1( pipe(pfd) )
    for (i = 0; i < 2; i++) {
        ec_neg1( flags = fcntl(pfd[i], F_GETFL) )
        if ((flags & O_ACCMODE) == O_RDONLY)
            printf("pfd[%d] O_RDONLY\n", i);
        if ((flags & O_ACCMODE) == O_WRONLY)
            printf("pfd[%d] O_WRONLY\n", i);
        if ((flags & O_ACCMODE) == O_RDWR)
            printf("pfd[%d] O_RDWR\n", i);
    }
    return;

EC_CLEANUP_BGN
    EC_FLUSH("pipe_access_mode")
EC_CLEANUP_END
}
```

On Linux I got what you would probably expect:

```
pfd[0] O_RDONLY
pfd[1] O_WRONLY
```

But look what FreeBSD and Solaris said:

```
pfd[0] O_RDWR
pfd[1] O_RDWR
```

Both ends of the pipe are opened for reading and writing! It turns out that on those two systems (and on others) anything written on `pfd[0]` can be read on `pfd[1]`, anything written on `pfd[1]` can be read on `pfd[0]`, and the data is never mixed up. In fact, on those systems a single pipe is a divided highway, with no collisions. This means that we can set up two-way communication with only one pipe because it's bidirectional.

To see this, here's a revised version of `edinvoke` from the last example in the previous section:

```
static FILE *sndfp, *rcvfp;

static bool edinvoke(void)
{
    int pfd[2];
    pid_t pid;

    ec_neg1( pipe(pfd) )
    switch (fork()) {
    case -1:
        EC_FAIL
    case 0:
        ec_neg1( dup2(pfd[0], STDIN_FILENO) )
        ec_neg1( dup2(pfd[0], STDOUT_FILENO) )
        ec_neg1( close(pfd[0]) )
        execlp("ed", "ed", "-", (char *)NULL);
        EC_FAIL
    }
    ec_null( sndfp = fdopen(pfd[1], "w") )
    ec_null( rcvfp = fdopen(pfd[1], "r") )
    return true;

EC_CLEANUP_BGN
    if (pid == 0) {
        EC_FLUSH("edinvoke");
        _exit(EXIT_FAILURE);
    }
    return false;
EC_CLEANUP_END
}
```

That's all I changed—the rest of the program is identical. It worked exactly as before, but with one pipe instead of two.

There are two disadvantages of using bidirectional pipes instead of using two pipes unidirectionally:

- They're nonstandard, and therefore nonportable.
- A writer who also has to read can't close the writing end of the pipe to simulate an end-of-file because it has only one end. This means, for instance, that the `fsort` example from the previous section can't be written to use only one pipe. (Try it if you don't believe me.)

So, it's best to assume that pipes are unidirectional and to use two of them if you need two-way communication.

Exercises

6.1. Using as many shells as you have access to (e.g., `sh`, `ksh`, `bash`, `csh`), try typing bad command lines to see what error message you get, if any, and what happens. Try these examples for starters:

```
echo abc > f1 > f2
echo def | cat < f1
echo ghi > f1 | cat
cat < f1 < f2
echo jkl | | cat
echo jki | > f1
```

6.2. In the function `gettoken` in Section 6.4, in state `INQUOTE`, when `getchar` is called, not much attention is paid to an end-of-file. Is this a problem? Why or why not?

6.3. Add parameter replacement (e.g., `echo $PATH`) to the shell in Section 6.4.

6.4. Add wild-cards (file-name generation) to the shell in Section 6.4. Use the standard function `glob` if you have it (most systems do). If you don't have it, you can restrict matching to the last component of path names (i.e., `ls */*.c` is illegal).

6.5. Add a `goto` built-in statement to the shell of Section 6.4. Can you implement it as an external command to be run as a child process? (Hint: Review open file descriptions in Section 2.2.3.)

6.6. Add an `if` statement to the shell of Section 6.4.

6.7. Design and implement a menu shell. Instead of prompting for commands, it displays a menu of choices and the user simply picks one. You'll need menus for command arguments, too. Rather than building in the details of various commands, the shell should take its information about commands and their arguments from a database.

6.8. Design and implement a windowing shell. Use Curses if you have access to it. Divide the terminal screen into two halves (which is easier—a horizontal or vertical split?). Run a shell child process in each half. The user presses a function key or a control key (Ctrl-w, say) to select the active window. A command running in the inactive window continues to output to the screen, but when input is requested it blocks until the user activates the window and types a response. A window scrolls when it gets full, and the scrolled-off lines are gone forever. Only line-oriented input and output are supported. Design questions: Should the standard output of a com-

mand go directly to the CRT, to a specially designed filter, or back to the windowing shell? What about input? Optional extras (ranging from hard to extremely hard): The user can scroll a window back to see what disappeared (up to some limit). Screen-oriented output can be handled, as well as line-oriented.

6.9. Write a program that writes its process-ID to the standard output and then reads a list of process-IDs from its standard input. If its own process-ID is on the list, it prints out the list (using file descriptor 2). Otherwise, it adds its process-ID to the list and writes the entire list to its standard output. Then it repeats. Arrange five processes running this program into a ring, and let them play awhile (this is a computer game for a computer to play by itself).

6.10. Using a scheme similar to that of the interactive search program in Section 6.5, write a front-end to the desk calculator `dc` that allows the user to enter expressions in infix notation (2 + 3) instead of postfix (2 3+). (This is what the `bc` command does.)

6.11. Reverse the last example in Section 6.3 so that `wc` is the parent of `who`.

6.12. If you're using FreeBSD, Darwin, or Linux, try to figure out why the search program in Section 6.6 doesn't work. Hint: It generates errors on purpose, which is always a problematical technique, and their version of `ed` doesn't like it. Is this a bug, or is it documented on the `man` page, or both? Is there any way to fix it?

6.13. Extend the program you wrote in Exercise 5.14 to include process attributes from Appendix A that are explained in this chapter.

This page intentionally left blank