UNIX Programming — Second Edition

Brian W. Kernighan

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper is an introduction to programming on the UNIX† system. The emphasis is on how to write programs that interface to the operating system, either directly or through the standard I/O library. The topics discussed include

- handling command arguments
- rudimentary I/O; the standard input and output
- the standard I/O library; file system access
- low-level I/O: open, read, write, close, seek
- processes: exec, fork, pipes
- signals — interrupts, etc.

There is also an appendix which describes the standard I/O library in detail.

1. INTRODUCTION

This paper describes how to write programs that interface with the UNIX operating system in a non-trivial way. This includes programs that use files by name, that use pipes, that invoke other commands as they run, or that attempt to catch interrupts and other signals during execution.

The document collects material which is scattered throughout several sections of *The UNIX Programmer's Manual* [1] for Version 7 UNIX. There is no attempt to be complete; only generally useful material is dealt with. It is assumed that you will be programming in C, so you must be able to read the language roughly up to the level of *The C Programming Language* [2]. Some of the material in sections 2 through 4 is based on topics covered more carefully there. You should also be familiar with UNIX itself at least to the level of *UNIX for Beginners* [3].

2. BASICS

2.1. Program Arguments

When a C program is run as a command, the arguments on the command line are made available to the function main as an argument count argc and an array argv of pointers to character strings that contain the arguments. By convention, argv[0] is the command name itself, so argc is always greater than 0.

The following program illustrates the mechanism: it simply echoes its arguments back to the terminal. (This is essentially the echo command.)

---

† UNIX is a Trademark of Bell Laboratories.

```
main(argc, argv)     /* echo arguments */
int argc;
char *argv[];
{
        int i;

        for (i = 1; i < argc; i++)
                printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

argv is a pointer to an array whose individual elements are pointers to arrays of characters; each is terminated by \0, so they can be treated as strings. The program starts by printing argv[1] and loops until it has printed them all.

The argument count and the arguments are parameters to main. If you want to keep them around so other routines can get at them, you must copy them to external variables.

2.2. The ‘‘Standard Input’’ and ‘‘Standard Output’’

The simplest input mechanism is to read the ‘‘standard input,’’ which is generally the user’s terminal. The function getchar returns the next input character each time it is called. A file may be substituted for the terminal by using the < convention: if prog uses getchar, then the command line

```
prog <file
```

causes prog to read file instead of the terminal. prog itself need know nothing about where its input is coming from. This is also true if the input comes from another program via the pipe mechanism:

```
otherprog | prog
```

provides the standard input for prog from the standard output of otherprog.

getchar returns the value EOF when it encounters the end of file (or an error) on whatever you are reading. The value of EOF is normally defined to be -1, but it is unwise to take any advantage of that knowledge. As will become clear shortly, this value is automatically defined for you when you compile a program, and need not be of any concern.

Similarly, putchar(c) puts the character c on the ‘‘standard output,’’ which is also by default the terminal. The output can be captured on a file by using >: if prog uses putchar,

```
prog >outfile
```

writes the standard output on outfile instead of the terminal. outfile is created if it doesn’t exist; if it already exists, its previous contents are overwritten. And a pipe can be used:

```
prog | otherprog
```

puts the standard output of prog into the standard input of otherprog.

The function printf, which formats output in various ways, uses the same mechanism as putchar does, so calls to printf and putchar may be intermixed in any order; the output will appear in the order of the calls.

Similarly, the function scanf provides for formatted input conversion; it will read the standard input and break it up into strings, numbers, etc., as desired. scanf uses the same mechanism as getchar, so calls to them may also be intermixed.

Many programs read only one input and write one output; for such programs I/O with getchar, putchar, scanf, and printf may be entirely adequate, and it is almost always enough to get started. This is particularly true if the UNIX pipe facility is used to connect the output of one program to the input of the next. For example, the following program strips out all ascii control characters from its input (except for newline and tab).

```
#include <stdio.h>

main()    /* ccstrip: strip non-graphic characters */
{
        int c;
        while ((c = getchar()) != EOF)
                if ((c >= ' ' && c < 0177) || c == '\t' || c == '\n')
                        putchar(c);
        exit(0);
}
```

The line

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

should appear at the beginning of each source file. It causes the C compiler to read a file (*/usr/include/stdio.h*) of standard routines and symbols that includes the definition of EOF.

If it is necessary to treat multiple files, you can use cat to collect the files for you:

```
cat file1 file2 ... | ccstrip >output
```

and thus avoid learning how to access files from a program. By the way, the call to exit at the end is not necessary to make the program work properly, but it assures that any caller of the program will see a normal termination status (conventionally 0) from the program when it completes. Section 6 discusses status returns in more detail.

3. THE STANDARD I/O LIBRARY

The ''Standard I/O Library'' is a collection of routines intended to provide efficient and portable I/O services for most C programs. The standard I/O library is available on each system that supports C, so programs that confine their system interactions to its facilities can be transported from one system to another essentially without change.

In this section, we will discuss the basics of the standard I/O library. The appendix contains a more complete description of its capabilities.

3.1. File Access

The programs written so far have all read the standard input and written the standard output, which we have assumed are magically pre-defined. The next step is to write a program that accesses a file that is *not* already connected to the program. One simple example is *wc*, which counts the lines, words and characters in a set of files. For instance, the command

```
wc x.c y.c
```

prints the number of lines, words and characters in x.c and y.c and the totals.

The question is how to arrange for the named files to be read — that is, how to connect the file system names to the I/O statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be *opened* by the standard library function fopen. fopen takes an external name (like x.c or y.c), does some housekeeping and negotiation with the operating system, and returns an internal name which must be used in subsequent reads or writes of the file.

This internal name is actually a pointer, called a *file pointer*, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don't need to know the details, because part of the standard I/O definitions obtained by including stdio.h is a structure definition called FILE. The only declaration needed for a file pointer is exemplified by

```
FILE    *fp, *fopen();
```

This says that fp is a pointer to a FILE, and fopen returns a pointer to a FILE. (FILE is a type name, like int, not a structure tag.

The actual call to fopen in a program is

```
fp = fopen(name, mode);
```

The first argument of fopen is the name of the file, as a character string. The second argument is the mode, also as a character string, which indicates how you intend to use the file. The only allowable modes are read ("r"), write ("w"), or append ("a").

If a file that you open for writing or appending does not exist, it is created (if possible). Opening an existing file for writing causes the old contents to be discarded. Trying to read a file that does not exist is an error, and there may be other causes of error as well (like trying to read a file when you don't have permission). If there is any error, fopen will return the null pointer value NULL (which is defined as zero in stdio.h).

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which getc and putc are the simplest. getc returns the next character from a file; it needs the file pointer to tell it what file. Thus

```
c = getc(fp)
```

places in c the next character from the file referred to by fp; it returns EOF when it reaches end of file. putc is the inverse of getc:

```
putc(c, fp)
```

puts the character c on the file fp and returns c. getc and putc return EOF on error.

When a program is started, three files are opened automatically, and file pointers are provided for them. These files are the standard input, the standard output, and the standard error output; the corresponding file pointers are called stdin, stdout, and stderr. Normally these are all connected to the terminal, but may be redirected to files or pipes as described in Section 2.2. stdin, stdout and stderr are pre-defined in the I/O library as the standard input, output and error files; they may be used anywhere an object of type FILE * can be. They are constants, however, *not* variables, so don't try to assign to them.

With some of the preliminaries out of the way, we can now write *wc*. The basic design is one that has been found convenient for many programs: if there are command-line arguments, they are processed in order. If there are no arguments, the standard input is processed. This way the program can be used stand-alone or as part of a larger process.

```
#include <stdio.h>

main(argc, argv)     /* wc: count lines, words, chars */
int argc;
char *argv[];
{
	int c, i, inword;
	FILE *fp, *fopen();
	long linect, wordct, charct;
	long tlinect = 0, twordct = 0, tcharct = 0;

	i = 1;
	fp = stdin;
	do {
		if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
			fprintf(stderr, "wc: can't open %s\n", argv[i]);
			continue;
		}
		linect = wordct = charct = inword = 0;
		while ((c = getc(fp)) != EOF) {
			charct++;
			if (c == '\n')
				linect++;
			if (c == ' ' || c == '\t' || c == '\n')
				inword = 0;
			else if (inword == 0) {
				inword = 1;
				wordct++;
			}
		}
		printf("%7ld %7ld %7ld", linect, wordct, charct);
		printf(argc > 1 ? " %s\n" : "\n", argv[i]);
		fclose(fp);
		tlinect += linect;
		twordct += wordct;
		tcharct += charct;
	} while (++i < argc);
	if (argc > 2)
		printf("%7ld %7ld %7ld total\n", tlinect, twordct, tcharct);
	exit(0);
}
```

The function fprintf is identical to printf, save that the first argument is a file pointer that specifies the file to be written.

The function fclose is the inverse of fopen; it breaks the connection between the file pointer and the external name that was established by fopen, freeing the file pointer for another file. Since there is a limit on the number of files that a program may have open simultaneously, it's a good idea to free things when they are no longer needed. There is also another reason to call fclose on an output file — it flushes the buffer in which putc is collecting output. (fclose is called automatically for each open file when a program terminates normally.)

3.2. Error Handling — Stderr and Exit

stderr is assigned to a program in the same way that stdin and stdout are. Output written on stderr appears on the user's terminal even if the standard output is redirected. *wc* writes its diagnostics on stderr instead of stdout so that if one of the files can't be accessed for some reason, the message finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program actually signals errors in another way, using the function exit to terminate program execution. The argument of exit is available to whatever process called it (see Section 6), so the success or failure of the program can be tested by another program that uses this one as a sub-process. By convention, a return value of 0 signals that all is well; non-zero values signal abnormal situations.

exit itself calls fclose for each open output file, to flush out any buffered output, then calls a routine named _exit. The function _exit causes immediate termination without any buffer flushing; it may be called directly if desired.

3.3. Miscellaneous I/O Functions

The standard I/O library provides several other I/O functions besides those we have illustrated above.

Normally output with putc, etc., is buffered (except to stderr); to force it out immediately, use fflush(fp).

fscanf is identical to scanf, except that its first argument is a file pointer (as with fprintf) that specifies the file from which the input comes; it returns EOF at end of file.

The functions sscanf and sprintf are identical to fscanf and fprintf, except that the first argument names a character string instead of a file pointer. The conversion is done from the string for sscanf and into it for sprintf.

fgets(buf, size, fp) copies the next line from fp, up to and including a newline, into buf; at most size-1 characters are copied; it returns NULL at end of file. fputs(buf, fp) writes the string in buf onto file fp.

The function ungetc(c, fp) ‘‘pushes back’’ the character c onto the input stream fp; a subsequent call to getc, fscanf, etc., will encounter c. Only one character of pushback per file is permitted.

4. LOW-LEVEL I/O

This section describes the bottom level of I/O on the UNIX system. The lowest level of I/O in UNIX provides no buffering or any other services; it is in fact a direct entry into the operating system. You are entirely on your own, but on the other hand, you have the most control over what happens. And since the calls and usage are quite simple, this isn’t as bad as it sounds.

4.1. File Descriptors

In the UNIX operating system, all input and output is done by reading or writing files, because all peripheral devices, even the user’s terminal, are files in the file system. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called ‘‘opening’’ the file. If you are going to write on a file, it may also be necessary to create it. The system checks your right to do so (Does the file exist? Do you have permission to access it?), and if all is well, returns a small positive integer called a *file descriptor.* Whenever I/O is to be done on the file, the file descriptor is used instead of the name to identify the file. (This is roughly analogous to the use of READ(5,...) and WRITE(6,...) in Fortran.) All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

The file pointers discussed in section 3 are similar in spirit to file descriptors, but file descriptors are more fundamental. A file pointer is a pointer to a structure that contains, among other things, the file descriptor for the file in question.

Since input and output involving the user’s terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the ‘‘shell’’) runs a program, it opens three files, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error output. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and writes file descriptors 1 and 2, it can do terminal I/O without worrying about opening the files.

If I/O is redirected to and from files with < and >, as in

```
prog <infile >outfile
```

the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Similar observations hold if the input or output is associated with a pipe. Normally file descriptor 2 remains attached to the terminal, so error messages can go there. In all cases, the file assignments are changed by the shell, not by the program. The program does not need to know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

4.2. Read and Write

All input and output is done by two functions called read and write. For both, the first argument is a file descriptor. The second argument is a buffer in your program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are

```
n_read = read(fd, buf, n);

n_written = write(fd, buf, n);
```

Each call returns a byte count which is the number of bytes actually transferred. On reading, the number of bytes returned may be less than the number asked for, because fewer than n bytes remained to be read. (When the file is a terminal, read normally reads only up to the next newline, which is generally less than what was requested.) A return value of zero bytes implies end of file, and -1 indicates an error of some sort. For writing, the returned value is the number of bytes actually written; it is generally an error if this isn't equal to the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two most common values are 1, which means one character at a time (''unbuffered''), and 512, which corresponds to a physical blocksize on many peripheral devices. This latter size will be most efficient, but even character at a time I/O is not inordinately expensive.

Putting these facts together, we can write a simple program to copy its input to its output. This program will copy anything to anything, since the input and output can be redirected to any file or device.

```
#define   BUFSIZE        512       /* best size for PDP-11 UNIX */

main()    /* copy input to output */
{
          char      buf[BUFSIZE];
          int       n;

          while ((n = read(0, buf, BUFSIZE)) > 0)
                    write(1, buf, n);
          exit(0);
}
```

If the file size is not a multiple of BUFSIZE, some read will return a smaller number of bytes to be written by write; the next call to read after that will return zero.

It is instructive to see how read and write can be used to construct higher level routines like getchar, putchar, etc. For example, here is a version of getchar which does unbuffered input.

```
#define   CMASK  0377      /* for making char's > 0 */

getchar() /* unbuffered single character input */
{
          char c;

          return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

c *must* be declared char, because read accepts a character pointer. The character being returned must be masked with 0377 to ensure that it is positive; otherwise sign extension may make it negative. (The constant 0377 is appropriate for the PDP-11 but not necessarily for other machines.)

The second version of getchar does input in big chunks, and hands out the characters one at a time.

```
#define   CMASK  0377      /* for making char's > 0 */
#define   BUFSIZE          512

getchar() /* buffered version */
{
          static char      buf[BUFSIZE];
          static char      *bufp = buf;
          static int  n = 0;

          if (n == 0) {    /* buffer is empty */
                n = read(0, buf, BUFSIZE);
                bufp = buf;
          }
          return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

4.3. Open, Creat, Close, Unlink

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system entry points for this, open and creat [sic].

open is rather like the fopen discussed in the previous section, except that instead of returning a file pointer, it returns a file descriptor, which is just an int.

```
int fd;

fd = open(name, rwmode);
```

As with fopen, the name argument is a character string corresponding to the external file name. The access mode argument is different, however: rwmode is 0 for read, 1 for write, and 2 for read and write access. open returns -1 if any error occurs; otherwise it returns a valid file descriptor.

It is an error to try to open a file that does not exist. The entry point creat is provided to create new files, or to re-write old ones.

```
fd = creat(name, pmode);
```

returns a file descriptor if it was able to create the file called name, and -1 if not. If the file already exists, creat will truncate it to zero length; it is not an error to creat a file that already exists.

If the file is brand new, creat creates it with the *protection mode* specified by the pmode argument. In the UNIX file system, there are nine bits of protection information associated with a file, controlling read, write and execute permission for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is most convenient for specifying the permissions. For example, 0755 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the UNIX utility *cp*, a program which copies one file to another. (The main simplification is that our version copies only one file, and does not permit the second argument to be a directory.)

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644 /* RW for owner, R for group, others */

main(argc, argv)	/* cp: copy f1 to f2 */
int argc;
char *argv[];
{
	int	f1, f2, n;
	char	buf[BUFSIZE];

	if (argc != 3)
		error("Usage: cp from to", NULL);
	if ((f1 = open(argv[1], 0)) == -1)
		error("cp: can't open %s", argv[1]);
	if ((f2 = creat(argv[2], PMODE)) == -1)
		error("cp: can't create %s", argv[2]);

	while ((n = read(f1, buf, BUFSIZE)) > 0)
		if (write(f2, buf, n) != n)
			error("cp: write error", NULL);
	exit(0);
}

error(s1, s2)	/* print error message and die */
char *s1, *s2;
{
	printf(s1, s2);
	printf("\n");
	exit(1);
}
```

As we said earlier, there is a limit (typically 15-25) on the number of files which a program may have open simultaneously. Accordingly, any program which intends to process many files must be prepared to re-use file descriptors. The routine close breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. Termination of a program via exit or return from the main program closes all open files.

The function unlink(filename) removes the file filename from the file system.

4.4. Random Access — Seek and Lseek

File I/O is normally sequential: each read or write takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call lseek provides a way to move around in a file without actually reading or writing:

```
lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is fd to move to position offset, which is taken relative to the location specified by origin. Subsequent reading or writing will begin at that position. offset is a long; fd and origin are int's. origin can be 0, 1, or 2 to specify that offset is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append to a file, seek to the end before writing:

```
lseek(fd, 0L, 2);
```

To get back to the beginning (‘‘rewind’’),

```
lseek(fd, 0L, 0);
```

Notice the 0L argument; it could also be written as (long) 0.

With lseek, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file.

```
get(fd, pos, buf, n) /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
	lseek(fd, pos, 0);	/* get to pos */
	return(read(fd, buf, n));
}
```

In pre-version 7 UNIX, the basic entry point to the I/O system is called seek. seek is identical to lseek, except that its offset argument is an int rather than a long. Accordingly, since PDP-11 integers have only 16 bits, the offset specified for seek is limited to 65,535; for this reason, origin values of 3, 4, 5 cause seek to multiply the given offset by 512 (the number of bytes in one physical block) and then interpret origin as if it were 0, 1, or 2 respectively. Thus to get to an arbitrary place in a large file requires two seeks, first one which selects the block, then one which has origin equal to 1 and moves to the desired byte within the block.

4.5. Error Processing

The routines discussed in this section, and in fact all the routines which are direct entries into the system can incur errors. Usually they indicate an error by returning a value of −1. Sometimes it is nice to know what sort of error occurred; for this purpose all these routines, when appropriate, leave an error number in the external cell errno. The meanings of the various error numbers are listed in the introduction to Section II of the *UNIX Programmer's Manual,* so your program can, for example, determine if an attempt to open a file failed because it did not exist or because the user lacked permission to read it. Perhaps more commonly, you may want to print out the reason for failure. The routine perror will print a message associated with the value of errno; more generally, sys_errno is an array of character strings which can be indexed by errno and printed by your program.

5. PROCESSES

It is often easier to use a program written by someone else than to invent one's own. This section describes how to execute a program from within another.

5.1. The ''System'' Function

The easiest way to execute a program from another is to use the standard library routine system. system takes one argument, a command string exactly as typed at the terminal (except for the newline at the end) and executes it. For instance, to time-stamp the output of a program,

```
main()
{
	system("date");
	/* rest of processing */
}
```

If the command string has to be built from pieces, the in-memory formatting capabilities of sprintf may be useful.

Remember than getc and putc normally buffer their input; terminal I/O will not be properly synchronized unless this buffering is defeated. For output, use fflush; for input, see setbuf in the appendix.

5.2. Low-Level Process Creation — Execl and Execv

If you're not using the standard library, or if you need finer control over what happens, you will have to construct calls to other programs using the more primitive routines that the standard library's system routine is based on.

The most basic operation is to execute another program *without returning*, by using the routine execl.

To print the date as the last action of a running program, use

```
execl("/bin/date", "date", NULL);
```

The first argument to execl is the *file name* of the command; you have to know where it is found in the file system. The second argument is conventionally the program name (that is, the last component of the file name), but this is seldom used except as a place-holder. If the command takes arguments, they are strung out after this; the end of the list is marked by a NULL argument.

The execl call overlays the existing program with the new one, runs that, then exits. There is *no* return to the original program.

More realistically, a program might fall into two or more phases that communicate only through temporary files. Here it is natural to make the second pass simply an execl call from the first.

The one exception to the rule that the original program never gets control back occurs when there is an error, for example if the file can't be found or is not executable. If you don't know where date is located, say

```
execl("/bin/date", "date", NULL);
execl("/usr/bin/date", "date", NULL);
fprintf(stderr, "Someone stole 'date'\n");
```

A variant of execl called execv is useful when you don't know in advance how many arguments there are going to be. The call is

```
execv(filename, argp);
```

where argp is an array of pointers to the arguments; the last pointer in the array must be NULL so execv can tell where the list ends. As with execl, filename is the file in which the program is found, and argp[0] is the name of the program. (This arrangement is identical to the argv array for program arguments.)

Neither of these routines provides the niceties of normal command execution. There is no automatic search of multiple directories — you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like <, >, *, ?, and [] in the argument list. If you want these, use execl to invoke the shell sh, which then does all the work. Construct a string commandline that contains the complete command as it would have been typed at the terminal, then say

```
execl("/bin/sh", "sh", "-c", commandline, NULL);
```

The shell is assumed to be at a fixed place, /bin/sh. Its argument -c says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in commandline.

5.3. Control of Processes — Fork and Wait

So far what we've talked about isn't really all that useful by itself. Now we will show how to regain control after running a program with execl or execv. Since these routines simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a routine called fork:

```
proc_id = fork();
```

splits the program into two copies, both of which continue to run. The only difference between the two is the value of proc_id, the ''process id.'' In one of these processes (the ''child''), proc_id is zero. In the other (the ''parent''), proc_id is non-zero; it is the process number of the child. Thus the basic way to call, and return from, another program is

```
if (fork() == 0)
        execl("/bin/sh", "sh", "-c", cmd, NULL);        /* in child */
```

And in fact, except for handling errors, this is sufficient. The fork makes two copies of the program. In the child, the value returned by fork is zero, so it calls execl which does the command and then dies. In the parent, fork returns non-zero so it skips the execl. (If there is any error, fork returns -1).

More often, the parent wants to wait for the child to terminate before continuing itself. This can be done with the function wait:

```
int status;

if (fork() == 0)
        execl(...);
wait(&status);
```

This still doesn't handle any abnormal conditions, such as a failure of the execl or fork, or the possibility that there might be more than one child running simultaneously. (The wait returns the process id of the terminated child, if you want to check it against the value returned by fork.) Finally, this fragment doesn't deal with any funny behavior on the part of the child (which is reported in status). Still, these three lines are the heart of the standard library's system routine, which we'll show in a moment.

The status returned by wait encodes in its low-order eight bits the system's idea of the child's termination status; it is 0 for normal termination and non-zero to indicate various kinds of problems. The next higher eight bits are taken from the argument of the call to exit which caused a normal termination of the child process. It is good coding practice for all programs to return meaningful status.

When a program is called by the shell, the three file descriptors 0, 1, and 2 are set up pointing at the right files, and all other possible file descriptors are available for use. When this program calls another one, correct etiquette suggests making sure the same conditions hold. Neither fork nor the exec calls affects open files in any way. If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the execl. Conversely, if a caller buffers an input stream, the called program will lose any information that has been read by the caller.

5.4. Pipes

A *pipe* is an I/O channel intended for use between two cooperating processes: one process writes into the pipe, while the other reads. The system looks after buffering the data and synchronizing the two processes. Most pipes are created by the shell, as in

```
ls | pr
```

which connects the standard output of ls to the standard input of pr. Sometimes, however, it is most convenient for a process to set up its own plumbing; in this section, we will illustrate how the pipe connection is established and used.

The system call pipe creates a pipe. Since a pipe is used for both reading and writing, two file descriptors are returned; the actual usage is like this:

```
int     fd[2];

stat = pipe(fd);
if (stat == -1)
        /* there was an error ... */
```

fd is an array of two file descriptors, where fd[0] is the read side of the pipe and fd[1] is for writing. These may be used in read, write and close calls just like any other file descriptors.

If a process reads a pipe which is empty, it will wait until data arrives; if a process writes into a pipe which is too full, it will wait until the pipe empties somewhat. If the write side of the pipe is closed, a subsequent read will encounter end of file.

To illustrate the use of pipes in a realistic setting, let us write a function called popen(cmd, mode), which creates a process cmd (just as system does), and returns a file descriptor that will either read or write that process, according to mode. That is, the call

```
fout = popen("pr", WRITE);
```

creates a process that executes the pr command; subsequent write calls using the file descriptor fout will send their data to that process through the pipe.

popen first creates the the pipe with a pipe system call; it then forks to create two copies of itself. The child decides whether it is supposed to read or write, closes the other side of the pipe, then calls the shell (via execl) to run the desired process. The parent likewise closes the end of the pipe it does not use. These closes are necessary to make end-of-file tests work properly. For example, if a child that intends to read fails to close the write end of the pipe, it will never see the end of the pipe file, just because there is one writer potentially active.

```
#include <stdio.h>

#define   READ     0
#define   WRITE    1
#define   tst(a, b)   (mode == READ ? (b) : (a))
static    int      popen_pid;

popen(cmd, mode)
char      *cmd;
int       mode;
{
          int p[2];

          if (pipe(p) < 0)
                    return(NULL);
          if ((popen_pid = fork()) == 0) {
                    close(tst(p[WRITE], p[READ]));
                    close(tst(0, 1));
                    dup(tst(p[READ], p[WRITE]));
                    close(tst(p[READ], p[WRITE]));
                    execl("/bin/sh", "sh", "-c", cmd, 0);
                    _exit(1);  /* disaster has occurred if we get here */
          }
          if (popen_pid == -1)
                    return(NULL);
          close(tst(p[READ], p[WRITE]));
          return(tst(p[WRITE], p[READ]));
}
```

The sequence of closes in the child is a bit tricky. Suppose that the task is to create a child process that will read data from the parent. Then the first close closes the write side of the pipe, leaving the read side open. The lines

```
close(tst(0, 1));
dup(tst(p[READ], p[WRITE]));
```

are the conventional way to associate the pipe descriptor with the standard input of the child. The close closes file descriptor 0, that is, the standard input. dup is a system call that returns a duplicate of an already open file descriptor. File descriptors are assigned in increasing order and the first available one is returned, so the effect of the dup is to copy the file descriptor for the pipe (read side) to file descriptor 0; thus the read side of the pipe becomes the standard input. (Yes, this is a bit tricky, but it's a standard idiom.) Finally, the old read side of the pipe is closed.

A similar sequence of operations takes place when the child process is supposed to write from the parent instead of reading. You may find it a useful exercise to step through that case.

The job is not quite done, for we still need a function pclose to close the pipe created by popen. The main reason for using a separate function rather than close is that it is desirable to wait for the termination of the child process. First, the return value from pclose indicates whether the process succeeded. Equally important when a process creates several children is that only a bounded number of unwaited-for children can exist, even if some of them have terminated; performing the wait lays the child to rest. Thus:

```
#include <signal.h>

pclose(fd)      /* close pipe fd */
int fd;
{
        register r, (*hstat)(), (*istat)(), (*qstat)();
        int     status;
        extern int popen_pid;

        close(fd);
        istat = signal(SIGINT, SIG_IGN);
        qstat = signal(SIGQUIT, SIG_IGN);
        hstat = signal(SIGHUP, SIG_IGN);
        while ((r = wait(&status)) != popen_pid && r != -1);
        if (r == -1)
                status = -1;
        signal(SIGINT, istat);
        signal(SIGQUIT, qstat);
        signal(SIGHUP, hstat);
        return(status);
}
```

The calls to signal make sure that no interrupts, etc., interfere with the waiting process; this is the topic of the next section.

The routine as written has the limitation that only one pipe may be open at once, because of the single shared variable popen_pid; it really should be an array indexed by file descriptor. A popen function, with slightly different arguments and return value is available as part of the standard I/O library discussed below. As currently written, it shares the same limitation.

6. SIGNALS — INTERRUPTS AND ALL THAT

This section is concerned with how to deal gracefully with signals from the outside world (like interrupts), and with program faults. Since there's nothing very useful that can be done from within C about program faults, which arise mainly from illegal memory references or from execution of peculiar instructions, we'll discuss only the outside-world signals: *interrupt*, which is sent when the DEL character is typed; *quit*, generated by the FS character; *hangup*, caused by hanging up the phone; and *terminate*, generated by the *kill* command. When one of these events occurs, the signal is sent to *all* processes which were started from the corresponding terminal; unless other arrangements have been made, the signal terminates the process. In the *quit* case, a core image file is written for debugging purposes.

The routine which alters the default action is called signal. It has two arguments: the first specifies the signal, and the second specifies how to treat it. The first argument is just a number code, but the second is the address is either a function, or a somewhat strange code that requests that the signal either be ignored, or that it be given the default action. The include file signal.h gives names for the various arguments, and should always be included when signals are used. Thus

```
#include <signal.h>
 ...
signal(SIGINT, SIG_IGN);
```

causes interrupts to be ignored, while

```
signal(SIGINT, SIG_DFL);
```

restores the default action of process termination. In all cases, signal returns the previous value of the signal. The second argument to signal may instead be the name of a function (which has to be declared explicitly if the compiler hasn't seen it already). In this case, the named routine will be called when the signal occurs. Most commonly this facility is used to allow the program to clean up unfinished business before terminating, for example to delete a temporary file:

```
#include <signal.h>

main()
{
        int onintr();

        if (signal(SIGINT, SIG_IGN) != SIG_IGN)
                signal(SIGINT, onintr);

        /* Process ... */

        exit(0);
}

onintr()
{
        unlink(tempfile);
        exit(1);
}
```

Why the test and the double call to signal? Recall that signals like interrupt are sent to *all* processes started from a particular terminal. Accordingly, when a program is to be run non-interactively (started by &), the shell turns off interrupts for it so it won't be stopped by interrupts intended for foreground processes. If this program began by announcing that all interrupts were to be sent to the onintr routine regardless, that would undo the shell's effort to protect it when run in the background.

The solution, shown above, is to test the state of interrupt handling, and to continue to ignore interrupts if they are already being ignored. The code as written depends on the fact that signal returns the previous state of a particular signal. If signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command-processing loop. Think of a text editor: interrupting a long printout should not cause it to terminate and lose the work already done. The outline of the code for this case is probably best written like this:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf sjbuf;

main()
{
        int (*istat)(), onintr();

        istat = signal(SIGINT, SIG_IGN);        /* save original status */
        setjmp(sjbuf);        /* save current stack position */
        if (istat != SIG_IGN)
                signal(SIGINT, onintr);

        /* main processing loop */
}

onintr()
{
        printf("\nInterrupt\n");
        longjmp(sjbuf);        /* return to saved state */
}
```

The include file setjmp.h declares the type jmp_buf an object in which the state can be saved. sjbuf is such an object; it is an array of some sort. The setjmp routine then saves the state of things. When an interrupt occurs, a call is forced to the onintr routine, which can print a message, set flags, or whatever. longjmp takes as argument an object stored into by setjmp, and restores control to the location after the call to setjmp, so control (and the stack level) will pop back to the place in the main routine where the signal is set up and the main loop entered. Notice, by the way, that the signal gets set again after an interrupt occurs. This is necessary; most signals are automatically reset to their default action when they occur.

Some programs that want to detect signals simply can't be stopped at an arbitrary point, for example in the middle of updating a linked list. If the routine called on occurrence of a signal sets a flag and then returns instead of calling exit or longjmp, execution will continue at the exact point it was interrupted. The interrupt flag can then be tested later.

There is one difficulty associated with this approach. Suppose the program is reading the terminal when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that ''execution resumes at the exact point it was interrupted,'' the program would continue reading the terminal until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading; he presumably would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the terminal read when execution resumes after the signal, returning an error code which indicates what happened.

Thus programs which catch and resume execution after signals should be prepared for ''errors'' which are caused by interrupted system calls. (The ones to watch out for are reads from a terminal, wait, and pause.) A program whose onintr program just sets intflag, resets the interrupt signal, and returns, should usually include code like the following when it reads the standard input:

```
if (getchar() == EOF)
        if (intflag)
                /* EOF caused by interrupt */
        else
                /* true end-of-file */
```

A final subtlety to keep in mind becomes important when signal-catching is combined with execution of other programs. Suppose a program catches interrupts, and also includes a method (like ''!'' in the editor) whereby other programs can be executed. Then the code should look something like this:

```
if (fork() == 0)
        execl(...);
signal(SIGINT, SIG_IGN);    /* ignore interrupts */
wait(&status);          /* until the child is done */
signal(SIGINT, onintr);         /* restore interrupts */
```

Why is this? Again, it's not obvious but not really difficult. Suppose the program you call catches its own interrupts. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read your terminal. But the calling program will also pop out of its wait for the subprogram and read your terminal. Having two processes reading your terminal is very unfortunate, since the system figuratively flips a coin to decide who should get each line of input. A simple way out is to have the parent program ignore interrupts until the child is done. This reasoning is reflected in the standard I/O library function system:

```
#include <signal.h>

system(s)	/* run command string s */
char *s;
{
	int status, pid, w;
	register int (*istat)(), (*qstat)();

	if ((pid = fork()) == 0) {
		execl("/bin/sh", "sh", "-c", s, 0);
		_exit(127);
	}
	istat = signal(SIGINT, SIG_IGN);
	qstat = signal(SIGQUIT, SIG_IGN);
	while ((w = wait(&status)) != pid && w != -1)
		;
	if (w == -1)
		status = -1;
	signal(SIGINT, istat);
	signal(SIGQUIT, qstat);
	return(status);
}
```

As an aside on declarations, the function signal obviously has a rather strange second argument. It is in fact a pointer to a function delivering an integer, and this is also the type of the signal routine itself. The two values SIG_IGN and SIG_DFL have the right type, but are chosen so they coincide with no possible actual functions. For the enthusiast, here is how they are defined for the PDP-11; the definitions should be sufficiently ugly and nonportable to encourage use of the include file.

```
#define	SIG_DFL	(int (*)())0
#define	SIG_IGN	(int (*)())1
```

References

[1] K. L. Thompson and D. M. Ritchie, *The UNIX Programmer's Manual*, Bell Laboratories, 1978.

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., 1978.

[3] B. W. Kernighan, ''UNIX for Beginners — Second Edition.'' Bell Laboratories, 1978.

Appendix — The Standard I/O Library

D. M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

The standard I/O library was designed with the following goals in mind.

1. It must be as efficient as possible, both in time and in space, so that there will be no hesitation in using it no matter how critical the application.
2. It must be simple to use, and also free of the magic numbers and mysterious calls whose use mars the understandability and portability of many programs using older packages.
3. The interface provided should be applicable on all machines, whether or not the programs which implement it are directly portable to other systems, or to machines other than the PDP-11 running a version of UNIX.

1. General Usage

Each program using the library must have the line

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

which defines certain macros and variables. The routines are in the normal C library, so no special library argument is needed for loading. All names in the include file intended only for internal use begin with an underscore _ to reduce the possibility of collision with a user name. The names intended to be visible outside the package are

stdin The name of the standard input file

stdout The name of the standard output file

stderr The name of the standard error file

EOF is actually −1, and is the value returned by the read routines on end-of-file or error.

NULL is a notation for the null pointer, returned by pointer-valued functions to indicate an error

FILE expands to struct _iob and is a useful shorthand when declaring pointers to streams.

BUFSIZ is a number (viz. 512) of the size suitable for an I/O buffer supplied by the user. See setbuf, below.

getc, getchar, putc, putchar, feof, ferror, fileno
are defined as macros. Their actions are described below; they are mentioned here to point out that it is not possible to redeclare them and that they are not actually functions; thus, for example, they may not have breakpoints set on them.

The routines in this package offer the convenience of automatic buffer allocation and output flushing where appropriate. The names stdin, stdout, and stderr are in effect constants and may not be assigned to.

2. Calls

FILE *fopen(filename, type) char *filename, *type;

opens the file and, if needed, allocates a buffer for it. filename is a character string specifying the name. type is a character string (not a single character). It may be "r", "w", or "a" to indicate intent to read, write, or append. The value returned is a file pointer. If it is NULL the attempt to open failed.

FILE *freopen(filename, type, ioptr) char *filename, *type; FILE *ioptr;

The stream named by ioptr is closed, if necessary, and then reopened as if by fopen. If the attempt to open fails, NULL is returned, otherwise ioptr, which will now refer to the new file. Often the reopened stream is stdin or stdout.

int getc(ioptr) FILE *ioptr;

returns the next character from the stream named by ioptr, which is a pointer to a file such as returned by fopen, or the name stdin. The integer EOF is returned on end-of-file or when an error occurs. The null character \0 is a legal character.

int fgetc(ioptr) FILE *ioptr;

acts like getc but is a genuine function, not a macro, so it can be pointed to, passed as an argument, etc.

putc(c, ioptr) FILE *ioptr;

putc writes the character c on the output stream named by ioptr, which is a value returned from fopen or perhaps stdout or stderr. The character is returned as value, but EOF is returned on error.

fputc(c, ioptr) FILE *ioptr;

acts like putc but is a genuine function, not a macro.

fclose(ioptr) FILE *ioptr;

The file corresponding to ioptr is closed after any buffers are emptied. A buffer allocated by the I/O system is freed. fclose is automatic on normal termination of the program.

fflush(ioptr) FILE *ioptr;

Any buffered information on the (output) stream named by ioptr is written out. Output files are normally buffered if and only if they are not directed to the terminal; however, stderr always starts off unbuffered and remains so unless setbuf is used, or unless it is reopened.

exit(errcode);

terminates the process and returns its argument as status to the parent. This is a special version of the routine which calls fflush for each output file. To terminate without flushing, use _exit.

feof(ioptr) FILE *ioptr;

returns non-zero when end-of-file has occurred on the specified input stream.

ferror(ioptr) FILE *ioptr;

returns non-zero when an error has occurred while reading or writing the named stream. The error indication lasts until the file has been closed.

getchar();

is identical to getc(stdin).

putchar(c);

is identical to putc(c, stdout).

char *fgets(s, n, ioptr) char *s; FILE *ioptr;

reads up to n-1 characters from the stream ioptr into the character pointer s. The read terminates with a newline character. The newline character is placed in the buffer followed by a null character. fgets returns the first argument, or NULL if error or end-of-file occurred.

fputs(s, ioptr) char *s; FILE *ioptr;

writes the null-terminated string (character array) s on the stream ioptr. No newline is appended. No value is returned.

ungetc(c, ioptr) FILE *ioptr;

The argument character c is pushed back on the input stream named by ioptr. Only one character may be pushed back.

printf(format, a1, ...) char *format;
fprintf(ioptr, format, a1, ...) FILE *ioptr; char *format;
sprintf(s, format, a1, ...)char *s, *format;

printf writes on the standard output. fprintf writes on the named output stream. sprintf puts characters in the character array (string) named by s. The specifications are as described in section printf(3) of the *UNIX Programmer's Manual.*

scanf(format, a1, ...) char *format;
fscanf(ioptr, format, a1, ...) FILE *ioptr; char *format;
sscanf(s, format, a1, ...) char *s, *format;

scanf reads from the standard input. fscanf reads from the named input stream. sscanf reads from the character string supplied as s. scanf reads characters, interprets them according to a format, and stores the results in its arguments. Each routine expects as arguments a control string format, and a set of arguments, *each of which must be a pointer,* indicating where the converted input should be stored.

scanf returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, EOF is returned; note that this is different from 0, which means that the next input character does not match what was called for in the control string.

fread(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;
: reads nitems of data beginning at ptr from file ioptr. No advance notification that binary I/O is being done is required; when, for portability reasons, it becomes required, it will be done by adding an additional character to the mode-string on the fopen call.

fwrite(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;
: Like fread, but in the other direction.

rewind(ioptr) FILE *ioptr;
: rewinds the stream named by ioptr. It is not very useful except on input, since a rewound output file is still open only for output.

system(string) char *string;
: The string is executed by the shell as if typed at the terminal.

getw(ioptr) FILE *ioptr;
: returns the next word from the input stream named by ioptr. EOF is returned on end-of-file or error, but since this a perfectly good integer feof and ferror should be used. A ‘‘word’’ is 16 bits on the PDP-11.

putw(w, ioptr) FILE *ioptr;
: writes the integer w on the named output stream.

setbuf(ioptr, buf) FILE *ioptr; char *buf;
: setbuf may be used after a stream has been opened but before I/O has started. If buf is NULL, the stream will be unbuffered. Otherwise the buffer supplied will be used. It must be a character array of sufficient size:

```
char    buf[BUFSIZ];
```

fileno(ioptr) FILE *ioptr;
: returns the integer file descriptor associated with the file.

fseek(ioptr, offset, ptrname) FILE *ioptr; long offset;
: The location of the next byte in the stream named by ioptr is adjusted. offset is a long integer. If ptrname is 0, the offset is measured from the beginning of the file; if ptrname is 1, the offset is measured from the current read or write pointer; if ptrname is 2, the offset is measured from the end of the file. The routine accounts properly for any buffering. (When this routine is used on non-UNIX systems, the offset must be a value returned from ftell and the ptrname must be 0).

long ftell(ioptr) FILE *ioptr;
: The byte offset, measured from the beginning of the file, associated with the named stream is returned. Any buffering is properly accounted for. (On non-UNIX systems the value of this call is useful only for handing to fseek, so as to position the file to the same place it was when ftell was called.)

getpw(uid, buf) char *buf;
: The password file is searched for the given integer user ID. If an appropriate line is found, it is copied into the character array buf, and 0 is returned. If no line is found corresponding to the user ID then 1 is returned.

char *malloc(num);
: allocates num bytes. The pointer returned is sufficiently well aligned to be usable for any purpose.

NULL is returned if no space is available.

char *calloc(num, size);

allocates space for num items each of size size. The space is guaranteed to be set to 0 and the pointer is sufficiently well aligned to be usable for any purpose. NULL is returned if no space is available .

cfree(ptr) char *ptr;

Space is returned to the pool used by calloc. Disorder can be expected if the pointer was not obtained from calloc.

The following are macros whose definitions may be obtained by including <ctype.h>.

isalpha(c) returns non-zero if the argument is alphabetic.

isupper(c) returns non-zero if the argument is upper-case alphabetic.

islower(c) returns non-zero if the argument is lower-case alphabetic.

isdigit(c) returns non-zero if the argument is a digit.

isspace(c) returns non-zero if the argument is a spacing character: tab, newline, carriage return, vertical tab, form feed, space.

ispunct(c) returns non-zero if the argument is any punctuation character, i.e., not a space, letter, digit or control character.

isalnum(c) returns non-zero if the argument is a letter or a digit.

isprint(c) returns non-zero if the argument is printable — a letter, digit, or punctuation character.

iscntrl(c) returns non-zero if the argument is a control character.

isascii(c) returns non-zero if the argument is an ascii character, i.e., less than octal 0200.

toupper(c) returns the upper-case character corresponding to the lower-case letter c.

tolower(c) returns the lower-case character corresponding to the upper-case letter c.