Input and Output in Unix and C

Basic Abstractions

Unix and C support a unified abstraction for input and output (I/O) known as *file descriptors*. Input and output from everything ranging from devices to files to network connections uses the same abstraction. In particular, the operating system maintains an array of structures with information about I/O channels, with each channel occupying one place in this table. The array index at which a given channel appears can thus be used to locate the corresponding information within the table, and a file descriptor is nothing more than an integer. Most operating systems limit the size of the table to 1,024 entries by default, so descriptors are typically in the range 0 to 1,023. A diagram appears in Figure 1.

Notice that the first three entries in the array of I/O channels are occupied by the "standard" I/O channels for a program. These channels are set up by the operating system before a program starts. If you execute a program by itself from within a shell, input comes from the keyboard, and output (both normal and error output) goes to the monitor. However, these defaults are easily overridden. In fact, you probably use a graphical window manager when working, in which case the output from your programs does not go to the monitor, but instead to the window manager for display in the window in which your program was started. In the original scheme for providing network services on Unix machines, known as `inetd` (for "Internet Daemon"), the operating system started programs in response to incoming network connections, replacing the standard input and output channels for the new program with the incoming connection. Network services could thus be written and tested easily from any standard shell, then simply redirected to accept input and send output across the network when they were ready.

The information in the I/O channel structure allows the operating system to differentiate between the different types of I/O channels as necessary, and this information can be accessed and manipulated by a wide array of generic and special-purpose system calls, but most of these are beyond the scope of our class. We will consider only a certain class of fairly general-purpose calls in this discussion.

In particular, we focus on the calls that use streams. A *stream* is a logical array of bytes that flow from one place to another through an I/O channel. Some types of I/O channels do not fit readily into the stream model; some network protocols, for example, break data into packets; most devices have control/status registers as well as data registers, and the access pattern necessary to control these devices is generally not the simple linear progression that the stream provides. Channels that can fit into the stream model include input from a keyboard, output to a monitor, files on a disk[1], and certain types of network protocols.

The stream abstraction also provides support for buffering part of the stream in order to improve performance. Files on disk are stored in blocks of four or eight kB, but can take tens of milliseconds to retrieve (tens of millions of processor clock cycles). If this delay were incurred for each byte read from a file, a program would run quite slowly. Even interacting with the operating system through a system call is relatively slow, however, often requiring tens or hundreds of thousands of cycles or more. Buffering reduces the number of interactions with the operating system by bringing data into the program in large blocks and using C library functions to handle most of the actual data transfer for a stream.

Buffering also helps to simplify the implementation of certain expected behaviors. For example, reading from the keyboard typically returns nothing until the user presses the RETURN/ENTER key. If this buffering is turned off, every application must process BACKSPACE, since a keystroke delivered to the application cannot otherwise be taken back. With the default buffering strategy, BACKSPACE is handled by keyboard-processing code, and application programs see only lines that have been completed by pressing RETURN.

[1] The disk itself is a block device, but the filesystem serves to translate this abstraction into a stream for any given file.

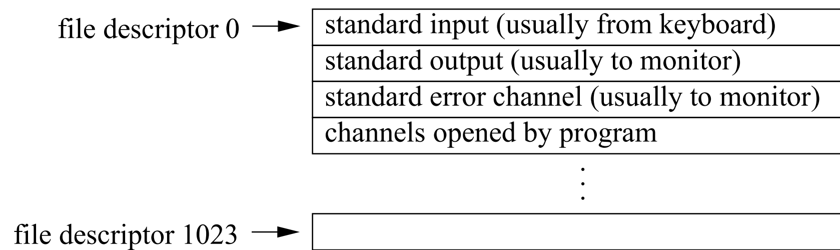

Figure 1: The array of I/O channel structures maintained for a program by the operating system. The array index for a channel's structure is used to identify the channel, and is passed around as a small integer known as a file descriptor.

In C, a stream is represented by a pointer to a structure containing information about the kind of buffering desired as well as the file descriptor to be used by the stream. The structure is written FILE (all capitals), and is allocated by library code, allowing a program to simply declare pointers to these structures in order to manipulate streams. For example:

```
FILE* my_file;
```

declares a variable to refer to a stream.

Default and New Streams

The three default file descriptors created by the operating system are also associated with streams before a program begins to execute. The input stream is named `stdin`, the output stream `stdout`, and the error stream `stderr`. Like the file descriptors, `stdin` can only be read, not written, while `stdout` and `stderr` can only be written, not read.

Other streams can be created with several functions, the simplest of which are those used to access the files stored on disk. As you should already be aware, Unix uses a hierarchical file system in which the names of files can consist of sequence of directory names followed by a name for the file within its local directory.[2] Directories correspond exactly to the folders used by graphical file system browsers. The file names used within a program are the same as those used within a Unix shell (a shell is just a program, after all); also like a shell, each program has a notion of a current directory, so files with no directory names in front of them refer to those files in the directory in which the program was started (assuming that the program has not changed its current directory).

The function below opens a file:

```
FILE* fopen (const char* file_name, const char* mode);
```

The first argument is a string containing the name of the file. The second argument is a string specifying what types of operations are to be performed on the new stream. The function returns NULL if the open fails, and the `perror` function can be called to print a human-readable error message in this case, such as "file not found" or "file not readable." If the file is opened successfully, the function returns a new stream, which should eventually be closed with `fclose`, as described below.

The mode consists of a letter followed by an optional plus sign. If the plus sign is included, the file is opened for both reading and writing. If the plus sign is not included, the file is opened for either reading or writing (depending on the letter, as we discuss shortly), but not both. The letter "r" is used to open an existing file; an error is returned if the file does not exist, and the file is opened for reading if no plus sign is included in the mode. The letter "w" creates a new file for writing, first deleting an existing file of the given name if it exists, and allowing only writing if no plus sign is included in the mode string. Finally, the letter "a" is used to write to the end of an existing file; in this case, if no file exists yet, a new one is created. With the append option, the default mode is writing, and writing begins at the end of the file.

[2] It may interest you to know that Unix "files" can actually be other channels in disguise, including everything from devices to network connections to channels to existing programs.

Finally, the letter “b” can optionally be included after the first letter (or the plus sign) with any mode, but has no effect. On some older systems, the “b” signified that binary data were being stored in the file, and that certain standard translations on ASCII text should not be performed to avoid corrupting the binary data. However, these translations are for the most part obsolete, and are never performed on Unix platforms.

When a program is done using a stream, it should close the stream to free up the I/O channel resource, of which a limited number are available for each program. For this purpose, use the following function:

```
int fclose (FILE* stream);
```

This function takes a stream and attempts to close it. If no errors occurred in accesses to the stream, the function returns 0. If an error occurred, it returns EOF (-1).

All streams are closed when a program finishes execution, but it is a bad habit to rely on program termination rather than using the `fclose` function. While you are unlikely to see any difference in this class, consider the impact of not closing network connections in a web server: after the first 1,021 connections, the server begins to drop all requests, as it has no free I/O channels.

Character by Character I/O

Our description of I/O functions now parallels the textbook, but we will cover a few more functions than are described in the text. We will also differentiate between the functions that operate on any stream and the shortcut functions that operate on `stdin` and `stdout`.

The functions below support reading and writing single characters to streams:

```
int fgetc (FILE* stream);  /* read one character (a function) */
int getc (FILE* stream);   /* read one character (a macro)    */
int fputc (int c, FILE* stream);  /* write one character (a function) */
int putc (int c, FILE* stream);   /* write one character (a macro)    */
```

The first two functions return the ASCII character read from the stream (converted into an integer), or EOF (-1) if the read attempt failed. These calls by default *block* until input is available. That is, a user’s not having typed a character yet does not cause the call to fail. Instead, the operating system puts the program to sleep until a character is typed. Recall that the same thing occurs in the LC-3 system calls, which wait for a character to be available rather than returning a failure indicator. Failure thus indicates conditions such as reaching the end of an input file.

The difference between `fgetc` and `getc` is that the first is a function and creates a function call in the assembly code generated by a compiler, while the second is a preprocessor macro-operation that copies the necessary code in place of the call before compilation. In older machines, these functions allowed a tradeoff between superior performance (`getc`) and reduced program size (`fgetc`). In more modern machines, you’re probably better off using `fgetc`.

The `fputc` and `putc` functions write a single character (specified by the first argument) to a stream. Although an integer is passed, only a single unsigned character is actually written. These functions both return the value written if successful, or EOF (-1) on failure.

Shortcut functions, both based the macro version of the functions above, are available for reading and writing single characters to `stdin` and `stdout`. These functions are declared as follows:

```
int getchar (void);   /* read one character from stdin (a macro) */
int putchar (int c);  /* write one character to stdout (a macro) */
```

The return values and argument are the same as for the corresponding previous functions, but the values of the stream parameters are implicit in each case.

Reading and Writing Lines

For text files, it is usually most convenient to work with a line at a time, reading each line into an array of characters and treating it as a string, or writing each line into an array of characters before sending it off to the file. The two functions used for these purposes with streams are:

```
char* fgets (char* s, int n, FILE* stream); /* read one line  */
int fputs (const char* s, FILE* stream);    /* write a string */
```

The `fgets` function reads one line of text from a stream into the array of characters given by its first argument. The second argument specifies the size of the array, and `fgets` also stops reading if it runs out of room. The function always appends an ASCII NUL character to terminate the string, so it reads at most `n`-1 characters from the stream. If a full line is read, the linefeed (LF, or "\n" in C) character is also written into the specified array of characters. The function returns its first argument when successful, and NULL when no further data are available from the stream (or some other error occurs).

The `fputs` function writes a string to a stream. No additional characters are sent, so the string must include a linefeed character at the end if it is to appear as a line in a text file. The function returns the number of characters written or EOF on failure.

Shortcut functions are available for both of these functions, but the shortcut for reading a line does not allow the caller to specify the length of the array, and thus poses a security hazard. **You should never use the `gets` function!** The majority of network attacks use strategies based on exactly this type of function, so you should simply never use it. The shortcut function used to write a string to `stdout` is declared as:

```
int puts (char* s);  /* write one line to stdout */
```

This function differs from `fputs` not only in implicitly writing to `stdout`, but also in that it also writes a linefeed character to `stdout` after writing the string passed. A string meant to become a single line of output must therefore NOT include a linefeed character at the end. The return value meanings are the same as for `fputs`.

Formatted I/O

You have already seen the `scanf` and `printf` functions used to translate between the ASCII text representing human-readable text and the binary forms understood by a computer. These two functions are simply the shortcut forms of the more general functions for reading and writing formatted data to streams:

```
int fscanf (FILE* stream, const char* fmt, ...);
int fprintf (FILE* stream, const char* fmt, ...);
```

The only difference between these functions and those already familiar to you is the need to include the stream as the first argument. With `scanf`, the `stdin` stream is used implicitly, while `printf` implicitly writes to `stdout`. Note that any information that should instead be delivered to `stderr` must use the more general form.

A third form of these functions is also useful, particularly in combination with the functions described in the previous section for reading lines and writing strings to streams:

```
int sscanf (const char* s, const char* fmt, ...);
int sprintf (char* s, const char* fmt, ...);
```

These functions read and write formatted data to strings (arrays of characters). Note that the printing function does not allow the caller to specify the size of the array, and can thus be attacked in certain cases. Be careful to allocate enough space for a printed string; you may prefer to use the `snprintf` function instead, but I'm not sure that it is required by the ANSI standard, thus you may need to write this function yourself for fully portable code.

Binary I/O

The last set of functions that we cover allow you to send binary data, such as the contents of an array, directly to and from a stream. While Unix does not perform any translations on bytes, none of the preceding functions allow you to transfer arbitrary sequences of bytes, a fact often overlooked by novice programmers. Pretending that an array of integers is a string does not generally work, for example, as any zero byte in the array ends the “string.”

Before describing the functions, we need to explain some possibly new types. To reflect growth in file and memory sizes, ANSI C actually uses a separate type to specify sizes in bytes, allowing this type to grow (to 64 bits, for example) without necessarily growing the size of an integer. This type is called `size_t`, but you can think of it as unsigned 32-bit integer, and on most systems it is just that.

A second type, a pointer to a null type (`void*`), is used to allow automatic conversions to and from any other pointer type. In particular, if the type of a parameter to a function is `void*`, any pointer type can be passed without causing the compiler to repond with warnings or errors

The functions are declared as follows:

```
size_t fread (void* ptr, size_t size, size_t n_items, FILE* stream);
size_t fwrite (const void* ptr, size_t size, size_t n_items, FILE* stream);
```

The list of arguments to both functions is essentially the same. The first argument is a pointer to the memory to be filled with bytes from the stream (in the case of `fread`) or from which bytes should be written to the stream (in the case of `fwrite`). The second argument specifies the size of items to be read or written, typically using the `sizeof()` built-in function, which is evaluated at compile time to the size of a type (or variable’s type) in bytes. The third argument specifies how many such items should be read or written, and the last argument gives the stream. Both functions return the number of items (NOT the number of bytes) read or written to the stream. Typically, a return value equal to the third argument indicates that the call was completely successful, but partial success or total failure is also possible, such as when a disk fills up or a user exceeds a disk quota.

No shortcut functions are available, as binary data are not normally delivered by or to human users.