Lecture 11 - File I/O (last updated: 10/13/22)

In the previous lecture, we encountered three channels with which programs can communicate with their environment: stdin, stdout, and stderr. They are declared in stdio.h as the following FILE pointers (i.e., their type is FILE *):

```
extern FILE *stdin;
extern FILE *stdout;
extern FILE *stderr;
```

These FILE pointers are the same abstraction used to represent the files we're familiar with (e.g., text and binary files on our file system containing data); the core interface for standard I/O and file I/O is the same. In this lecture, we will explore that interface.

FILE pointers

The following declaration says, "stdin is a pointer to a FILE":

```
extern FILE *stdin;
```

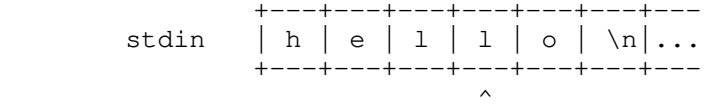
FILE is an "opaque type"; we are not supposed to know how exactly it is defined, only that it exists (as required by the C standard). It is usually defined with a typedef of some sort, e.g.:

```
typedef struct _IO_FILE FILE;
```

But the point of this interface is that we are not supposed to know what the fields of struct _IO_FILE are, or even that a FILE is a struct at all! Instead, we interact with FILE pointers through standard library functions, which know about and manipulate FILEs internally.

stdin, stdout, and stderr aren't the only FILE pointers you will encounter in C. We can also obtain and pass around FILE pointers to other files. Exactly what object the FILE pointer points to is not important; it just acts as a "handle" to something---a "file"---from the environment which we can communicate with.

What is important is what a FILE pointer represents: here, a "file" means a stream of bytes, i.e., a sequence of bytes with a position. For example, if we typed in "hello\n" to stdin, but we have only read 3 bytes, the file stream might look something like this:



Files (streams of bytes) differs from memory (an array of bytes) in that a stream is supposed to be accessed in sequence; e.g., if I read another 'l' from stdin, I will advance the position ("^"), and the next byte I read will be 'o'. In contrast, memory does not prescribe any such access order.

Though FILE pointers all share similar file stream semantics, specific FILE pointers may have capabilities that others do not. For example, some FILE pointers can only be read from (e.g., stdin); others may only be written to (e.g., stdout, stderr); some allow you to change the file stream position, while others do not.

Opening and closing files

stdin, stdout, and stderr are already available to all C programs, but to get a FILE pointer corresponding to any other file, we use fopen() ("file open"):

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

fopen() returns a non-NULL FILE pointer on success, and returns NULL on failure.

For example:

```
FILE *fp = fopen("myfile.txt", "r");
```

Both arguments are strings; the first argument is the path of the file we are trying to open, while the second is the "mode" we are trying to open this file in. Here are valid values for the mode argument (as documented in man fopen):

- "r": open file for reading; fail if file does not already exist. The stream is positioned at the beginning of the file.
- "w": open file for writing; create file if it does not exist, and overwrite it (i.e., delete old contents) if it already does. The stream is positioned at the beginning of the file.
- "a": open file for appending (writing at the end of the file); create file if it does not exist, but keep old contents if it already does. The stream is positioned at the end of the file.
- "r+": open file for reading and writing; fail if file does not already exist. The stream is positioned at the beginning of the file.
- "w+": open file for reading and writing; create file if it does not exist, and overwrite it (i.e., delete old contents) if it already does. The stream is positioned at the beginning of the file.
- "a+": open file for reading and appending (writing at the end of the file); create file if it does not exist, but keep old contents if it already does. The initial position of the stream is unspecified.

You can add "b" to each of these modes (e.g., "wb", "r+b") to specify that you want to open the file in "binary" mode, though this doesn't actually do anything on Linux (it only makes a difference on some operating systems like Windows).

Once you are done using a FILE, you should always fclose() it, e.g.:

```
fclose(fp);
```

You only need to fclose() FILE pointers you obtain from fopen(); you do not need to fclose() any of the standard FILEs. After you fclose() a FILE pointer, you should no longer use it (similar principle as a dangling pointer).

Writing output

To write to any FILE pointer, you can use fprintf():

```
    fprintf(stdout, "write to stdout\n");
    fprintf(stderr, "write to stderr\n");
    fprintf(fp, "write to myfile.txt\n");
```

If you do not need string formatting (see end of this lecture note), you can also use fputs() to write a string, or fputc() to write a single character:

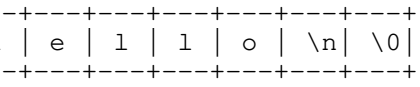
```
    fputs("write to stdout\n", stdout);
    fputs("write to stderr\n", stderr);
    fputs("write to myfile.txt\n", fp);

    fputc('o', stdout);
    fputc('e', stderr);
    fputc('t', fp);
```

The first argument to fputs() has type char *; it is supposed to be a string, i.e., a pointer to the beginning of a null-terminated char array. fputs() will keep outputting characters until it reaches the null byte, but it will NOT output the null byte itself.

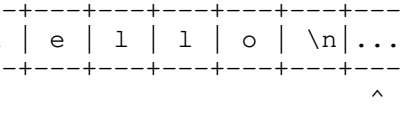
So if you call:

```
    char *s = "hello\n"
    fputs(s, stderr);
```

s points to the following null-terminatred character array in memory:



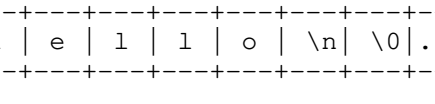
But only the following bytes are written to stderr:



If you would like to write arbitrary bytes out to a file stream, including null bytes, you should use fwrite():

```
    fwrite(s, 1, 7, stderr);
```

fwrite() expects you to tell it how much to write out in terms of "items". The second parameter indicates the size of each item, while the third parameter indicates the number of items; the first parameter is a pointer to the first element of an array of those items. So in the above call, we are asking fwrite() to write 7 items, each 1 byte, from s, to stderr:



The difference between fputs() and fwrite() is that fwrite() allows you to explicitly convey the number of bytes you would like to write, without relying on a null byte to indicate when you would like to stop writing bytes. fputs() is usually only used to output textual data from strings, whereas fwrite() is used to output arbitrary binary data.

Note that you do NOT need to null-terminate text files with null bytes. Null bytes are usually only necessary for terminating C strings in memory, so that the length of each C string does not need to be stored elsewhere.

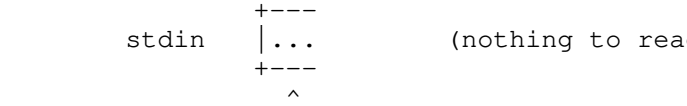
Reading input

Reading input is more complicated than writing output because of blocking. Blocking happens when you ask the operating system for something it cannot return immediately, so it pauses the execution of your process until it is ready to "unblock". From the perspective of your program, it will seem as if the function call does not return until your program unblocks.

An example of blocking is sleep():

```
sleep(1);                       // Blocks until 1 second later.
```

When you read input that is not yet available, e.g., read from stdin before you've typed in anything, your program will block until it receives input:



So if you call fgets() on stdin, it will block and your program will hang (not do anything) until you type something.

You can read string input using fgets() (opposite of fputs()), characters using fgetc() (opposite of fputc()), or binary input data using fread() (opposite of fwrite()).

fgetc() is the simplest: it just returns the character it read:

```
char c = fgetc(fp);
```

fgets() and fread() work by reading into a buffer you pass them:

```
char buf[1024];                 // Define 1024 bytes of space to read into

fgets(buf, sizeof(buf), fp);    // Read at most 1023 bytes to buf; add '\0'

fread(buf, 1, sizeof(buf), fp); // Read at most 1024 bytes to buf (no '\0')
```

Like with fputs() and fwrite(), the difference between fget() and fread() is between what kind of data of data you expect to read in, and how much you want to read in. fgets() expects textual data, and will unblock either when it encounters a newline, or when its buffer fills up; it guarantees that it will null-terminate its buffer. Meanwhile, fread() is better suited for arbitrary binary data, and only unblocks when its buffer fills up; it does not null-terminate its buffer, but returns the number of items it read in.

For reasons beyond the scope of this course, your terminal will not actually

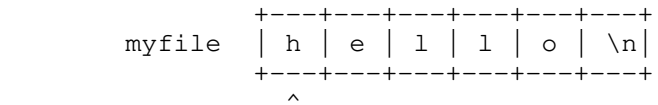
send anything to stdin until you type enter; in other words, reading from stdin will block until you start a new line. Note that this is only superficially related to the FILE output buffering discussed later in this lecture.

End of file

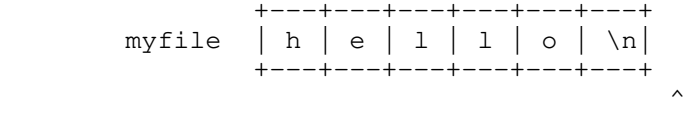
File reading functions like fgetc(), fgets(), and fread() will also unblock when they encounter some kind of error, or when they encounter an EOF ("end of file") condition. An EOF condition is not considered an error: it just happens when you try to read bytes, and the system tells you there cannot possibly be more bytes to read. For example, EOF happens when you try to read the 7th byte of a file that only contains 6 characters, e.g., one created like this:

```
$ echo "hello" > myfile         # myfile is 6 bytes (including newline)
```

When we fopen() myfile, we get a handle to this stream:



If we read all six bytes, we will advance the cursor past the end of the stream:



If we try to read from myfile again, we will encounter an EOF condition, and fgetc()/fgets()/fread() will unblock immediately. The return value of each of these functions will differ from normal:

- fgetc() normally returns a char of the byte it read; when it encounters EOF or an error, it returns special EOF value. Note that the EOF value is more than 1 byte! This is why fgetc() returns int instead of char. If you want to check for EOF, save the return value in an int instead of char:

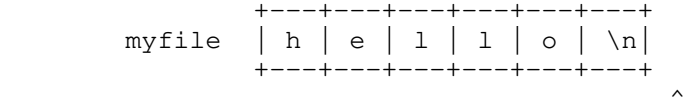
  ```
  int c = fgetc(fp);
  if (c == EOF)
      /* ... */
  ```

- fgets() normally returns a pointer to the buffer it writes to; when it encounters EOF or an error, it will return a NULL pointer.

- fread() normally returns the number of items it read; when it encounters EOF or an error, it returns less than the number of items you asked for

To distinguish between EOF and other errors, you should check using feof() or ferror().

Note that EOF is only encountered when we try to read PAST the end of a stream. If the stream position is here:



feof() will still return zero for the FILE pointer of myfile until we actually try to read more from myfile.

EOF conditions are not exclusive to real files; you can also encounter EOF on the stdin file stream. You can raise the EOF condition on EOF by typing Ctrl-D when there is no other pending input. (By the way, you can also use Ctrl-D to exit the shell; when you raise EOF in the shell, you are signaling the end of your shell session, and it will log you out.)

FILE output buffering

You may have noticed that printf() doesn't seem to do anything until you print a newline:

```
printf("hello...");
sleep(3);
prinf("world\n");
```

The above program will not print "hello..." until after it sleeps; that is, "hello..." and "world" will appear to print at the same time.

This surprising behavior is due to FILE buffering. printf() (and other FILE output functions like fputc(), fputs(), and fwrite()) will actually buffer bytes given to stdout, and wait until they encounter a newline character to transmit the output to the terminal. This is called "line buffering"; stdout is line buffered by default in order to batch output requests to the operating system are batched, which is often more efficient.

In contrast, stderr is "unbuffered", meaning it will always send output to the operating system immediately---usually useful for error messages and debugging. So the following will behave as expected:

```
fprintf(stderr, "hello...");
sleep(3);
fprinf(stderr, "world\n");
```

All other files are block-buffered by default; instead of waiting for a newline, output is buffered until a certain buffer size is reached.

If you insist on printing to stdout immediately, you can manually "flush" a FILE buffer using fflush() (this also works on block-buffered FILE pointers):

```
printf("hello...");
fflush(stdout);                 // Immediately flushes out "hello..."
sleep(3);
prinf("world\n");
```

Finally, if you want to turn off all buffering for a FILE pointer, and make it unbuffered like stderr, you can use setbuf() passing NULL as the second argument, e.g.:

```
setbuf(stdout, NULL);
```

Keep in mind that this may hamper performance when you are writing many bytes.

File seeking

Some FILE pointers allow you to adjust the stream position. For example, this is perfectly acceptable if the FILE pointer represents a normal file. You can use fseek() to do so, e.g.,:

```
fseek(fp, 0, SEEK_SET);         // Set stream position to beginning
fseek(fp, 5, SEEK_CUR);         // Advance stream position by 5
fseek(fp, -3, SEEK_CUR);        // Retract stream position by 3
fseek(fp, 12, SEEK_SET);        // Set stream position to 12 past beginning
fseek(fp, 0, SEEK_END);         // Set stream position to end of file
```

However, this only works on certain FILE pointers. For example, this does not make sense for stdout, since you cannot "unprint" what has already been printed.

String formatting

We've previously seen printf(), fprintf(), which write output to stdout or a FILE pointer according to a format string (e.g., "%d" or "%p (%s)\n"). This is what their signature looks like:

```
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

The ellipsis ("...") in the arguments is C syntax that means that these functions can take zero or more arguments after the format string.

scanf() and fscanf(), which read input from stdout or a FILE pointer according to a format string, have similar signatures:

```
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
```

If you want to use the string-formatting functionality without reading from or writing to a FILE, you can also use the "s"-variants of these functions:

- int sscanf(const char *input_string, const char *format, ...);

  Parse input from input_string rather than stdin

- int sprintf(char *output_buffer, const char *format, ...)

  Write output to output_buffer instead of stdout (output_buffer should point to sufficiently large memory)

- int snprintf(char *output_buffer, size_t size, const char *format, ...)

  A safer version of sprintf().

Summary

By now, we've encountered a lot of C standard library functions, especially for file I/O. It sometimes can be helpful to see them all in one place and learn them in pairs:

- `FILE *fopen(const char *pathname, const char *mode);`

  vs

  `int fclose(FILE *stream);`

- `char *gets(char *s);` // Do NOT use this; it is unsafe
  `char *fgets(char *s, int size, FILE *stream);`

  vs

  `int puts(const char *s);`
  `int fputs(const char *s, FILe *stream);`

- `int getchar(void);`
  `int fgetc(FILE *stream);`

  vs

  `int putchar(int c);`
  `int fputc(int c, FILE *stream);`

- `size_t fread(const void *ptr, size_t size, size_t n, FILE *stream);`

  vs

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

- `int printf(const char *format, ...);`
  `int fprintf(FILE *stream, const char *format, ...);`

  vs

  `int scanf(const char *format, ...);`
  `int fscanf(FILE *stream, const char *format, ...);`

It can also be helpful to know that C standard library functions tend to follow similar naming conventions for single-character abbreviations:

- 'f' at the beginning stands for "FILE pointer," e.g., fgets(), fprintf(), fseek(); these functions take a FILE pointer as an argument

- 's' stands for "string," e.g., fgets(), fputs(), snprintf(); these functions use or produce C strings, i.e., null-terminated arrays of characters

- 'f' at the end stands for "format," e.g., printf(), fprintf(), snprintf(); these functions require a format string of some kind

- 'n' somewhere usually means that the string function normally works with C strings, expects some explicit limit of n bytes, e.g., snprintf(), strncpy(), strncat()