Lecture 12 - File Contents and Descriptors (last updated: 10/19/22)

We previously saw how we can write bytes of data to and from files. In this lecture, we will consider details about the contents of those files, and compare FILE pointers with a lower-level interface specific to Linux/UNIX-like systems.

Dumping file contents

We can use text editors like Vim and shell utilities like cat to inspect file contents, but these tools only work well for viewing text, where the bytes correspond to ASCII characters that are meant to be displayed in a terminal. However, these tools don't help us inspect the contents of binary files, whose bytes aren't generally interpreted as ASCII characters, and may contain bytes outside of the printable ASCII character range (1--127).

We can of course write our own program to view these files. For example, the following code snippet prints each byte from fp to stdout, each on its own line and formatted in hexadecimal notation:

```
int b;
while ((b = fgetc(fp)) != EOF)
    printf("%x\n", b);
```

There are also many tools that essentially do this, including:

- od (short for "octal dump")
- hexdump
- xxd (short for "hex dump")

They all do the same thing, but have different feature sets. I prefer xxd because I like its default output format. For example, xxd shows us the contents of myadd.c byte-by-byte:

```
$ xxd myadd.c
00000000: 2369 6e63 6c75 6465 2022 6d79 6164 642e  #include "myadd.
00000010: 6822 0a0a 696e 7420 6164 6428 696e 7420  h"..int add(int
00000020: 782c 2069 6e74 2079 2920 7b0a 2020 2020  x, int y) {.
00000030: 7265 7475 726e 2078 202b 2079 3b0a 7d0a  return x + y;.}.
```

The left column represent the byte offset of each row, in hexdecimal notation. The middle columns show the file contents, 2 bytes (4 hex digits) at a time. The right column prints out those contents as ASCII, when they are printable.

We can do the same thing with myadd.o. xxd shows us that this object files contains a lot of things that aren't human-readable; here's the first 10 lines:

```
$ xxd myadd.o | head
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0100 3e00 0100 0000 0000 0000 0000 0000  ..>.............
00000020: 0000 0000 0000 0000 2806 0000 0000 0000  ........(.......
00000030: 0000 0000 4000 0000 0000 4000 1500 1400  ....@.....@.....
00000040: f30f 1efa 5548 89e5 897d fc89 75f8 8b55  ....UH...}..u..U
00000050: fc8b 45f8 01d0 5dc3 6b00 0000 0500 0108  ..E...].k.......
00000060: 0000 0000 0200 0000 001d 0000 0000 0000  ................
00000070: 0000 0000 0000 0000 0000 1800 0000 0000  ................
00000080: 0000 0000 0000 0361 6464 0001 0305 6700  .......add....g.
00000090: 0000 0000 0000 0000 0000 1800 0000 0000  ................
```

The column on the right tells us that most of the bytes in myadd.o cannot be interpreted as ASCII characters, and those that can don't make much sense.

Endianness

When we use fwrite() to write arbitrary data to a file, we need to be aware of what format those bytes appear in. In particular, when we write signed binary integers to a file, we should know what order those bytes will be written in. For instance, consider the x-filer program, which writes an integer to "x-file":

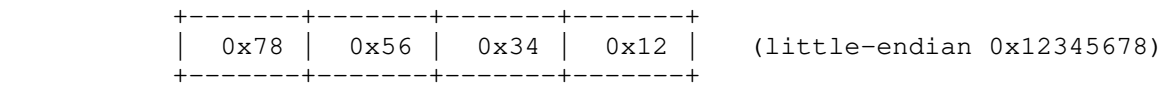
```
FILE *fp = fopen("x-file", "w");
int x = 0x12345678;
fwrite(&x, sizeof(x), 1, fp);
fclose(fp);
```

Here, x is an integer that consists of four bytes: 0x12, 0x34, 0x45, and 0x78. But what order do those bytes appear in memory, and consequently, in x-file?

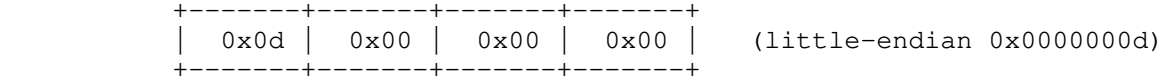
The order that each number's bytes appear in memory is known as "endianness". The endianness depends on what computer architecture your program is running on. On x86 (what we use on CLAC), they appear in reverse order:

```
$ ./x-filer             # write 0x12345678 to x-file
$ xxd x-file            # inspect contents of x-file
xxd x-file
00000000: 7856 3412                                xV4.
```

This endianness is known as "little-endian," where the least significant byte comes first. We read out a little-endian integer a byte at a time, they appear in reverse order. So in memory, x contains the following bytes:

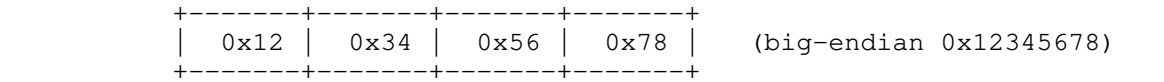


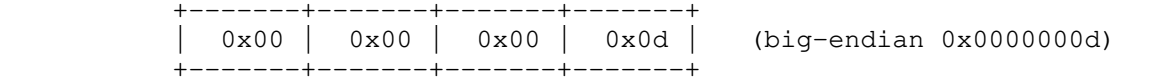
If x were a smaller number, e.g., 13 (i.e., 0xd), it would appear in memory as:



The least significant byte, 0x0d, comes first.

On a "big-endian" system, the bytes are arranged from most significant to least significant. So the above two values would appear as:





Most computers we use nowadays are little-endian; Arm processors, used in

smartphones and Apple computers, support both little- and big-endian, but are usually configured to run in little-endian mode.

File descriptors vs FILE pointers

FILE pointers are an abstraction for file streams provided by the C standard. FILE pointers are designed to work across different operating systems: they should work the same way on Windows as they do on Linux. However, Linux, and other Unix-like systems provide a lower-level file I/O interface via "file descriptors."

The core interface of file descriptors works very similarly to FILE pointers, with slight differences (see the man pages for details):

Type	FILE *fp	int fd

FILE pointers are implemented using pointers to the opaque FILE type; file descriptors are just integers.

FILE pointers work the same way on all operating systems where you can run C code. File descriptors are specific to POSIX systems; POSIX is a standard that describes UNIX-like systems.

Since file descriptors are just integers, they can also be easily represented in other programming languages running on POSIX systems, not just C, since most programming languages have integers. FILE pointers are specific to C (and C++ by extension).

Standard streams	stdin, stdout, stderr	0, 1, 2

The standard file streams, stdin, stdout, and stderr are also available via file descriptors 0, 1, and 2, respectively.

Opening files	fopen()	open()

FILE pointers are returned by fopen(), which takes two strings that specify the file path and the file mode; file descriptors are returned by open(), which takes a string specifying the file path and two integers specifying some flags and the file mode.

The flags argument allows files to be opened using POSIX- and Linux-specific options.

Writing bytes	fwrite()	write()

We writes bytes to FILE pointers using fwrite(), and to file descriptors using write().

When we write data using fwrite(), we tell it the number of items we want to write from our buffer, and the size of each item. When we want to write data using write(), we just tell it the number of bytes we want to write from our buffer.

For FILE pointers and fwrite(), output is (by default) line buffered for stdout, unbuffered for stderr, and block buffered for regular files. For file descriptors and write(), output is not visibly buffered (though this may depend on the file system you are writing to).

Reading bytes	fread()	read()

We read bytes from FILE pointers using fread(), and from file descriptors using read().

When we read data using fread(), we tell it the maximum number of items we want to read, and the size of each item; it returns the number of items read. When read data using read(), we just tell it the maximum number of bytes we want to read; it returns the number of bytes read.

fread() blocks until until it has read all the items we ask for, and only unblocks when we have read all items. It will only unblock with fewer items read than we asked for when it encounters an EOF condition or an error. read() is more finnicky, and may unblock with fewer bytes read than we requested, even without encountering errors or EOF. When read() encounters EOF, it returns 0; when read() encounters errors, it returns -1.

Closing files	fclose()	close()

FILE pointers are closed using fclose(); file descriptors are closed using close().

FILE pointers provide a simpler interface, and work across different operating systems. On POSIX systems, FILE pointers are implemented using file descriptors under the hood. File descriptors provide access to POSIX-specific features, most of which are beyond the scope of this course.

You are not expected to know the file descriptor API in any amount of detail, but you should recognize the similarities to and differences from FILE pointers.