Lecture 14 - Endianness (last updated: 2023-10-26)

Inspecting binary files

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 files, where the bytes correspond to human-readable ASCII characters. You can certainly a open binary file in Vim or cat the file to the terminal screen, but it would be hard to make out what the bytes are. (If you cat a binary file to the screen, by the way, the screen color may change or your font will switch to something you don't recognize if the binary file happens to contain the exact sequence of bytes that are meant to control the terminal settings. You can simply type "reset" and press ENTER, whether you can see what you are typing or not.)

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 a number of tools that essentially do this:

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

They are all very similar, but have different features and output formats. Here is an output from xxd, showing 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 a binary file, myadd.o for example. Here we pipe the output to the head command to display only 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  ................
```

In the right column, we see that xxd displays non-printable bytes as dots.

Endianness

Consider the following program, which writes an integer to a file:

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

We write the four bytes that make up the integer 0x12345678, which are 0x12, 0x34, 0x56, and 0x78. After we ran the program, we would expect that the file would contain the four bytes, 0x12, 0x34, 0x45, and 0x78, in that order.

Surprisingly, on most computers, you will see the following result:

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

fwrite() wrote the four bytes containing the number 0x12345678, but we see that the bytes have been written in reverse. fwrite() did not change anything; it wrote the four bytes of the integer x, from the first byte to the last byte.

This means, on the machine that the above result was produced, the integer 0x12345678 was actually stored in memory like this:

```
    +-------+-------+-------+-------+
    |  0x78 |  0x56 |  0x34 |  0x12 |   (0x12345678 in memory, little-endian)
    +-------+-------+-------+-------+
```

The way integers are stored in memory depends on the architecture of the computer that the program is running on. The result shown above was produced in what is called a "little-endian" system. In a little-endian system, integers are stored in reverse byte order -- the least significant byte comes first and the most significant byte comes last.

If x were a smaller number, say 3 for example, a little-endian system would store it like this:

```
    +-------+-------+-------+-------+
    |  0x03 |  0x00 |  0x00 |  0x00 |   (0x00000003 in memory, little-endian)
    +-------+-------+-------+-------+
```

Note that the reverse ordering is byte-by-byte, not bit-by-bit. That is, the bits for 0x00000003 are stored like this:

```
    00000011 00000000 00000000 00000000  (0x00000003 in memory, little-endian)
```

and NOT like this:

```
    11000000 00000000 00000000 00000000  (this is 0xc0000000, NOT 0x00000003)
```

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

```
    +-------+-------+-------+-------+
    | 0x12  | 0x34  | 0x56  | 0x78  |   (0x12345678 in memory, big-endian)
    +-------+-------+-------+-------+

    +-------+-------+-------+-------+
    | 0x00  | 0x00  | 0x00  | 0x03  |   (0x00000003 in memory, big-endian)
    +-------+-------+-------+-------+
```

Do not confuse the physical layout of an integer in memory with what the numerical value of the number is. Logically, the most significant byte of 0x12345678 is still 0x12 and the least significant byte is still 0x78, regardless of whether you are in a little-endian or big-endian system. The following piece of code:

```
    unsigned int n = 0x12345678;

    printf("Logically: (Most Significant Byte) "
            "%.2x %.2x %.2x %.2x (Least Significant Byte)\n",
            (n >> 24) & 0xff, (n >> 16) & 0xff, (n >> 8) & 0xff, n & 0xff);

    char *p = (char *)&n;
    printf("Physically in memory: (First Byte) "
            "%.2x %.2x %.2x %.2x (Last Byte)\n",
            p[0], p[1], p[2], p[3]);
```

will produce the following output in a little-endian system:

```
    Logically: (Most Significant Byte) 12 34 56 78 (Least Significant Byte)
    Physically in memory: (First Byte) 78 56 34 12 (Last Byte)
```

Most computers we use nowadays are little-endian. Computers with Intel’s x86 family of CPUs are all little-endian. ARM-based CPUs, used in smartphones and Apple computers, support both little- and big-endian, but are usually configured to run in little-endian mode.

Network byte order

When we wrote the integer 0x12345678 out to a file from a little-endian machine, we ended up with a file containing 0x78, 0x56, 0x34, and 0x12, in that order. What would happen if you were to open the file and read the four bytes into an integer as follows?

```
    FILE *f = fopen("x-file", "rb");
    assert(f);
    unsigned int n;
    fread(&n, sizeof(n), 1, f);
```

If we ran it in a little-endian machine, we would retrieve the original number 0x12345678. But if we ran it in a big-endian machine, we would read the bytes as the number 0x78563412, which is a completely different number.

In an environment where computers with different endianness need to exchange binary data, there must be an agreement on whether a sequence of bytes representing an integer should be interpreted as big-endian or little-endian.

One such environment is the Internet. When computers exchange information on the Internet, a crucial piece of the information is a 4-byte integer called an IP address which represents the location of a computer. The Internet Protocol specifies that the IP address (and other protocol data in integers) must be read in big-endian. Big-endian byte order is therefore also called the "network byte order".

This means that, a little-endian system must convert its integers to network byte order before it sends them over the Internet, and vice versa after it receives them. The following functions (or macros) will perform the necessary conversions:

```
    uint32_t htonl(uint32_t hostlong);

    uint16_t htons(uint16_t hostshort);

    uint32_t ntohl(uint32_t netlong);

    uint16_t ntohs(uint16_t netshort);
```

'h' means "host", 'n' means "network", 'l' is for a 32-bit integer (i.e., an int), and 's' is for a 16-bit integer (i.e., a short). So htonl() will take an int in host byte order and convert it into a network byte order.

The host byte order means the byte order of the system that you are running on. That is, the host byte order is little-endian in a little-endian system, and big-endian in a big-endian system. So htonl() does nothing on a big-endian system because the host byte order is already big-endian, which is the network byte order. The intention is that a program will always call these functions when it needs to write or read in network byte order, so that the program will work in any endian system without modifying the code. The program simply needs to be rebuilt.