HW2.5. Memory Endianness

Recall that the endianness of a system decides how multibyte blocks are stored in memory. In a little-endian system, memory is stored with the most significant byte at the highest address, while a big-endian system stores the most significant byte at the lowest address. Under normal circumstances, the endianness of a system is irrelevant, since we write and read to a variable in a consistent manner; if we write `int x = 5;`, we don't really care how the 5 is stored, as long as we get back `5` whenever we read that int.

There are two major cases when endianness matters:

- When sending data over a network. In particular, most network protocols use big-endian due to various physical effects, while most CPUs run in little-endian.
- When interpreting blocks of one size as another size (for example, interpreting an int as a char array). In this case, the order of the bytes will affect how our data gets interpreted in the new size. A later exercise will provide an example of when this might happen.

Suppose that we have the following bytes stored in memory:

```
0x00001000: 0x00
0x00001001: 0x01
0x00001002: 0xFF
0x00001003: 0xFF
```

We then run the line `printf("%d\n", *(int*)(0x1000));`. Assume that `int`s are 32 bits long (i.e. 4-bytes). Remember that `%d` will print out an integer not hex.

Q1.1: What gets printed out on a big-endian system?

131071 ✓ 100%

Q1.2: What gets printed out on a little-endian system? Hint: C uses 2's complement for signed integers, as with most signed number systems.

-65280 ✓ 100%

For this part, assume that we have a little endian system. We run the following code:

```
uint32_t *x = malloc(sizeof(uint32_t)*4); //Assume that x == 0x20000000
x[0] = 0xDEADBEEF;
x[1] = 0xC561C156;
x[2] = 0x00DC1A55;
x[3] = 0xABADCAFE;
uint64_t y = *((uint64_t*)x);
```

Q2.1: What byte is stored at memory address `0x2000000E`? Answer in hexadecimal, with no prefix.

0x AD ✓ 100%

Q2.2: What would `strlen((char*)x)` return?

11 ✓ 100%

Q2.3: What is the value of y? Answer in hexadecimal, with no prefix.

0x C561C156DEADBEEF ✓ 100%

Try a new variant

Correct answer

Q1.1: What gets printed out on a big-endian system?
131071

Q1.2: What gets printed out on a little-endian system? Hint: C uses 2's complement for signed integers, as with most signed number systems.
−65280

For this part, assume that we have a little endian system. We run the following code:

```
uint32_t *x = malloc(sizeof(uint32_t)*4); //Assume that x == 0x20000000
x[0] = 0xDEADBEEF;
x[1] = 0xC561C156;
x[2] = 0x00DC1A55;
x[3] = 0xABADCAFE;
uint64_t y = *((uint64_t*)x);
```

Q2.1: What byte is stored at memory address `0x2000000E`? Answer in hexadecimal, with no prefix.
0x `ad`

Q2.2: What would `strlen((char*)x)` return?
11

Q2.3: What is the value of y? Answer in hexadecimal, with no prefix.
0x `c561c156deadbeef`

Q1.1: In a big-endian system, the 32-bit number is `0x0001FFFF`. This is equal to 2^17-1, or 131072-1, or 131071.

Q1.2: In a little-endian system, the 32-bit number is `0xFFFF0100`. Note that this is actually a negative number! In particular, this is `0x00000100 - 0x10000 = 256 - 65536 = -65280`

Note: The arithmetic done in this question is slightly harder than what we expect from this class. Still, it is important to memorize at least the first 10 powers of 2, and useful to know up to the 16th power of 2 or so. At the very least, this acts as a checksum where you can verify a calculator answer is within the ballpark of what you expect.

Q2: Note that since we have a little-endian system, these integers are stored in little-endian. For this question, I will use the convention of adding a "0x" prefix before every "block" of data.

Since our data is `0x20000000: 0xDEADBEEF 0xC561C156 0x00DC1A55 0xABADCAFE`, the bytes stored in memory are:

`0x20000000: 0xEF 0xBE 0xAD 0xDE 0x56 0xC1 0x61 0xC5 0x55 0x1A 0xDC 0x00 0xFE 0xCA 0xAD 0xAB`

Q2.1: The byte at address `0x2000000E` is `0xAD` (the 15th byte)

Q2.2: `strlen` goes to the first `0x00` byte, which is the 12th byte. Our string length is thus 11.

Q2.3: We interpret the first 8 bytes as a little-endian integer, so we get `0xC561C156DEADBEEF`. Note that this is the same as x[1] and x[0] in order. This is because we reversed each 32-bit number once, then reversed it a second time when interpreting as a 64-bit number. In general, it is always possible to disassemble to the byte level and reconstruct to the new size, but the math still works correctly if you disassemble down to the GCF of the start and goal block size.

Submitted answer 3 **correct: 100%**
Submitted at 2022-09-03 08:54:21 (PDT)

hide

Q1.1: What gets printed out on a big-endian system?
131071 ✓ 100%

Q1.2: What gets printed out on a little-endian system? Hint: C uses 2's complement for signed integers, as with most signed number systems.

-65280 ✓ 100%

For this part, assume that we have a little endian system. We run the following code:

```
uint32_t *x = malloc(sizeof(uint32_t)*4); //Assume that x == 0x20000000
x[0] = 0xDEADBEEF;
x[1] = 0xC561C156;
x[2] = 0x00DC1A55;
x[3] = 0xABADCAFE;
uint64_t y = *((uint64_t*)x);
```

Q2.1: What byte is stored at memory address `0x2000000E`? Answer in hexadecimal, with no prefix.

0x AD ✓ 100%

Q2.2: What would `strlen((char*)x)` return?

11 ✓ 100%

Q2.3: What is the value of y? Answer in hexadecimal, with no prefix.

0x C561C156DEADBEEF ✓ 100%

Submitted answer 2 **partially correct: 80%**

Submitted at 2022-09-03 08:53:37 (PDT)

show

Submitted answer 1 **partially correct: 80%**

Submitted at 2022-09-03 08:52:21 (PDT)

show