CS 521: Systems Programming

Bit Manipulation

Lecture 19

Bit Manipulation

- Last class, we *lightly* discussed using the raw bits from a number to determine the mining “difficulty”
  - In C, we deal with numbers at the bit level quite a **bit**
    - (ha! 💩)
- They are also frequently used with **flags** to toggle options and combine them with other options
- These are called **bit fields**

Bit Fields

- You’ve already used bit fields
    - Yep! That’s right! In two different ways!
- `open(file, O_WRONLY | O_TRUNC | O_CREAT, 0666);`
    - Here, we are doing a bitwise `OR` to combine these fields
    - Write only, truncate, and create are all turned on
- They are also supported with struct members
    - Since the layout of a struct varies this is often emulated using a single integer

Another Example: Game Controller

- From Wikipedia:

```
/* Each of these preprocessor directives defines a single bit, corresponding
 * to one button on the controller. Button order matches that of the
 * Nintendo Entertainment System. */

#define KEY_RIGHT    (1 << 0)  /* 00000001 */
#define KEY_LEFT     (1 << 1)  /* 00000010 */
#define KEY_DOWN     (1 << 2)  /* 00000100 */
#define KEY_UP       (1 << 3)  /* 00001000 */
#define KEY_START    (1 << 4)  /* 00010000 */
#define KEY_SELECT   (1 << 5)  /* 00100000 */
#define KEY_B        (1 << 6)  /* 01000000 */
#define KEY_A        (1 << 7)  /* 10000000 */
```

- `<<` is the left shift operator; we can also shift to the right: `>>`

Writing in Binary

The previous example, written directly using `0b` syntax:

```
#define KEY_RIGHT  0b00000001
#define KEY_LEFT   0b00000010
#define KEY_DOWN   0b00000100
#define KEY_UP     0b00001000
#define KEY_START  0b00010000
#define KEY_SELECT 0b00100000
#define KEY_B      0b01000000
#define KEY_A      0b10000000
```

Bitwise Operators

- AND (`&`)
- OR (`|`)
- NOT (`~`)
- XOR (`^`)
- Bit shifting:
  - `>>`
  - `<<`

Bitwise AND

Compare the two sets of bits. If both bits are set, the result is a `1`:

```
    0101 (decimal 5)
AND 0011 (decimal 3)
  = 0001 (decimal 1)

/* C: */
0101 & 0011 = 0001
```

Often used to determine (test) if particular bits are set.

Bitwise OR

If either bit is set to 1, then the result is 1:

```
   0101 (decimal 5)
OR 0011 (decimal 3)
 = 0111 (decimal 7)

/* C: */
0101 | 0011 = 0111
```

Often used to set (turn on) particular bits.

Bitwise NOT

Flips the bits:

```
NOT 0111  (decimal 7)
  = 1000  (decimal 8)

/* C: */
~0111 = 1000
```

Bitwise XOR

Set to 1 if only one of the bits is 1, but set to 0 if both bits are 0 or both are 1:

```
    0101 (decimal 5)
XOR 0011 (decimal 3)
  = 0110 (decimal 6)

/* C: */
0101 ^ 0011 = 0110
```

Often used for toggling particular bits.

Back to our Game Controller

```
int gameControllerStatus = 0;

/* Sets the gameControllerStatus using OR */
void keyPressed(int key) {
     gameControllerStatus |= key;
}

/* Toggles the gameControllerStatus using XOR */
void keyPressed(int key) {
     gameControllerStatus ^= key;
}

/* Tests whether a bit is set using AND */
int isPressed(int key) {
    return gameControllerStatus & key;
}
```

Flipping Bits

- Want to toggle a flag?
  - `opts = opts ^ flag`
- Turn it off?
  - `opts = opts & ~flag`
- On?
  - `opts = opts | flag`

Shifting

You can move bits around with `<<` and `>>`:

```
00010111 << 1 = 00101110
00010111 << 3 = 10111000

00010111 >> 1 = 01001011
00010111 >> 3 = 00000010
```

Neat: A left shift by n is the same as multiplying by 2^n

Hexadecimal

- We use Base 10 for our daily lives
- Computers? Base 2
- And then there’s Base 16… Hexadecimal
  - Denoted by 0x
- Hexadecimal is a compact way to represent 4 bits of information
  - 4 bits = nibble
  - 8 bits = byte
- So `0xFF` gives us a byte’s worth of information

Hex Notation

- You might've noticed we've been using hexadecimal a lot when working with binary
  - `0-9` : 0-9 in binary
  - `A-F` : 10-15
  - So, we can store 16 bits of information
- Hex is nice when working with binary numbers:
  - `int i = 2815;`
  - `int i = 0xAFF;`
    - `0xAFF` = `1010 1111 1111`

The Difficulty Mask

- In P4, we start out with a difficulty mask of

  `0x00000FFF`

- Five 0’s and 3 F’s, or in binary:
  - 5 * 4 = 20 bits of zeros
  - 3 * 4 = 12 bits of ones

Setting Specific Bits

- Let’s say I asked you to set the 3rd bit in a bit field
- How would you accomplish this?
- `bit_field = bit_field | (0x1 << 3)`
- We can extend this approach to adjust the difficulty of our bitcoin miner
- We’ll just need to find out how many bits we need to to set to 1