S1 - Memory, Data and Addressing

Date: 2013-04-30
tags: hwswint

Performance

Performance is not just about CPU clock speed. Data and instructions reside in the memory and they must be fetched to the CPU from memory to operate upon.

CPU<->Memory bandwidth can bottleneck performance. Two things can be done for this:

- Increase memory bandwidth so that more data can travel at a time. (DDR1 -> DDR2 -> DDR3)
- Move less data into/out of the CPU. This can be achieved by putting a small amount of memory on the CPU itself (which is called cache memory).

Binary Representations

0s and 1s are represented by high and low voltage. It takes a little bit of time while changing the voltage and it is what limits the speed of the computing system. Electronics are designed to only care about the two specific voltages for 0 and 1 and not care about the voltages in between.

Representing information as bits and bytes

We're going to group our binary digits into groups of eight which are called bytes. The range can be-

00000000_2 - 11111111_2

In decimal, the above range can be expressed as-

0_{10} - 255_{10}

But using binary form to represent data in our programs can get tedious so hexadecimal form is used. Each hexadecimal digit is 4 bit long. A byte can also be represented as two hexadecimal digits. The range can be-

00_{16} - FF_{16} (which is also same as 0_{10} - 225_{10})

In C, $FA1D37B_{16}$ can be represented as `0xFA1D37B` or `0xfa1d37b`. This is an 8 digit hexadecimal number, so it is 8 x 4 bit = 32 bits or 4 bytes long number.

Byte Oriented Memory Organization

Memory is organized in bytes. Basically it is a big long array of bytes. Each byte has an address.

Machine Words

Machine has a "word size".

Until recently, most machines used 32-bit (4-byte) words. It limited address to 4GB (2^32 bytes) and it has become too small for memory intensive applications.

Now, most x86 systems use 64-bit (8-byte) words which has potential address space of 2^{64} ~ 1.8 x 10^{19} bytes (18EB - exabytes).

For backward compatibility, many CPUs support different word sizes of 16-bit, 8-bit, 4-bit, 2-bit and 1-bit.

Word Oriented Memory Organization

Addresses specify location of bytes in memory and each byte has an address.

In 32-bit systems, 4 bytes have to be grouped together into a word. And 64-bit systems have 8 byte words. So, what address do we give to those word with multiple bytes?

To maintain uniformity, address of a word is said to be the address of its first byte. [Refer to the last slide in `lecture_slides_01_012-memorg.pdf`]

Pointer is a data object that contains an address.

Byte Ordering

Say you want to store the 4-byte word `0xaabbccdd`. In what order will the bytes be stored? There are two different conventions for that.

Big Endian and Little Endian. (Origin: Gulliver's Travels)

Big Endian: The most significant byte of the number goes the lowest address.

Little Endian: The least significant byte of the number goes the lowest address.

Example -

Variable has 4-byte representation `0x01234567` and address of variable is `0x100`.

Big Endian

0x100	0x101	0x102	0x103
01	23	45	67

Little Endian

0x100	0x101	0x102	0x103
67	45	23	01

x86 architecture uses little endian convention.

Address and Pointers in C

Variable declaration: `int x, y;`

The two variables will find two locations in memory in which to store 2 integers (1 word each).

Pointer declaration: `int *ptr;`

Declares a variable `ptr` that is a pointer to a data item that in an integer. This will store an address rather than a value.

Assignment to a pointer: `ptr = &x;`

This assigns ptr to point to the address where x is located. `&` is used to get the address of a variable.

Dereference operator (*) is used to get the value pointed to by a pointer. `*ptr` will give us the value at the memory address given by the value of `ptr`.

Examples

- If `ptr = &x;` then `y = *ptr + 1` is same as `y = x + 1`.

- `*(&y)` is equivalent to y.

We can do arithmetic on pointers.

`ptr = ptr + 1` - Since type of ptr is int and an int uses 4 bytes, C automatically adds (1 x 4 =) 4. But this can be dangerous if we don't exactly know what is present at the next memory address.

Assignment in C

Left-Hand-Side = Right-Hand-Side

LHS must evaluate to a memory location (variable).

RHS must evaluate to a value (could be an address).

`int x, y; x = y + 3;` - Get value at y, add 3, put it in x

`int *x; int y; x = &y + 3;` - Get address of y, add (3 x 4 =) 12 to it, put it in x

`*x = y;` - Here `*` says to the compiler not to use `x` itself as the variable rather get the value stored at x, interpret it as an address, put value of y at that address.

Arrays

Arrays represent adjacent locations in memory that store same type of data objects. E.g. `int big_array[128];` allocates 512 adjacent bytes (size of int- 4 bytes x 128) in memory.

```
/* Lets assume that array starts at 0x00ff0000 */
int *array_ptr;
int big_array[128];
array_ptr = big_array; /* 0x00ff0000 */
array_ptr = &big_array[0]; /* 0x00ff0000 */
array_ptr = &big_array[0] + 3; /* 0x00ff000c (adds 3 * size of int) */
array_ptr = &big_array[3]; /* 0x00ff000c (adds 3 * size of int) */
array_ptr = big_array + 3; /* 0x00ff000c (adds 3 * size of int) */
*array_ptr = *array_ptr + 1; /* 0x00ff000c (but big_array[3] is incremented) */
```

For `array_ptr = big_array;`, when C sees that we are trying to assign an array to a different variable that is a pointer, it automatically assigns the memory address of the array.

The last one is a bit complicated. Lets first see RHS. `*array_ptr` gets the value pointed to by the pointer `array_ptr` which is `big_array[3]` as seen in second-last line. Now in LHS, `*array_ptr` says to go to the location pointed to by `array_ptr` which is the address of `big_array[3]`. So, in effect, it is equivalent to psuedo-code `big_array[3] = big_array[3] + 1`.

`array_ptr = &big_array[130];` - The array was only 128 element long but we are asking for index 130. But C doesn't give a fuck. It applies the same arithmetic calculation (adding 130 * size of int) and gives back address `0x00ff0208`. Beware of this!

In general, `&big_array[i]` is same as `(big_array + i)`, which implicitly computes `&big_array[0] + i * sizeof(big_array[0]);`

Representing strings

A C-style string is represented by an array of bytes. Elements are one-byte ASCII codes for each character. A 0 byte marks the end of the array.

`char S[4] = "lola";`

Getting the address of an integer in the memory.

```
void show_bytes(char *start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, *(start+i));
    printf("\n");
}
void show_int (int x) {
    show_bytes((char *) &x, sizeof(int));
}
```

First argument for `show_bytes` function is the address of the starting location in memory. It is a pointer pointing to the character (which are of size one byte). Second argument is the length of bytes we want to print. `%p` is for printing pointer and `%x` is for printing a value as hex and `%.2x` is for printing the value as two digit hex.

In `show_int` function, while passing the first parameter, `&x` is address of an integer, but by `(char *) &x`, we are casting it as an address of character.

Boolean Algebra

Encode "True" as 1 and "False" as 0.

- AND: A&B = 1 when both A is 1 and B is 1
- OR: A|B = 1 when either A is 1 or B is 1 or both
- XOR: A^B = 1 when either A is 1 or B is 1, but not both
- NOT: ~A = 1 when A is 0 and vice-versa

DeMorgan's Law: `~(A|B) = ~A & ~B`

Bitwise Operations

Bitwise operators &, |, ^, ~ are available in C. They can be applied to any "integral" data types (long, int, short, char). Operations are applied bitwise.

Examples:

```
char a, b, c;
a = (char)0x41; /* 0x41 -> 01000001 */
b = ~a;         /* 10111110 -> 0xBE */
a = (char)0;    /* 0x00 -> 00000000 */
b = ~a;         /* 11111111 -> 0xFF */
a = (char)0x69; /* 0x69 -> 01101001 */
b = (char)0x55; /* 0x55 -> 01010101 */
c = a & b;      /* 01000001 -> 0x41 */
```

Logic Operations

Logical operators in C: &&, || and !. `0` is viewed as false and any non-zero value is treated as true. Early termination occurs where possible.

Examples:

- `!0x41 --> 0x00`
- `!0x00 --> 0x01`
- `0x69 && 0x55 --> 0x01`

- `0x00 && 0x55 --> 0x00`
- `0x69 || 0x55 --> 0x01`
- `p && *p++` (avoids null pointer access `0x00000000`)

In the last one, if p is a null pointer (false), then the and operation will short-circuit and *p++ won't be executed. It is short for `if (p) { *p++; }`.

Representing & Manipulating Sets

Bit vectors can be used to represent sets.

Width w bit vector represents of {0,...,w-1}

a_j = 1 if j in A - each bit in the vector represents the absence (0) or presence (1) of an element in the set.

01101001

76543210

The set here is {0, 3, 5, 6}.

01010101

76543210

And the set here is {0, 2, 4, 6}.

Operations:

- & Intersections --> 01000001 {0, 6}
- | Union --> 01111101 {0, 2, 3, 4, 5, 6}
- ^ Symmetric difference --> 00111100 {2, 3, 4, 5}
- ~ Complement --> 10101010 {1, 3, 5, 7}