Introduction to Computer Organization

DIS 1H – Week 3

Slides modified from UT WANG

Logistics

Extra office hours – **Week 4**
Monday 2:00 P.M. – 4:00 P.M.
and
Friday 12:00 P.M. – 2:00 P.M.

Week 5 – No discussion on May 5, 2017

Agenda

- Switch statement
- Arrays and Structures
- Procedures
- Midterm review
- Arrays example
- More Struct and Union
- gdb

Practice Problem 3.37

Consider the following source code, where M and N are constants declared with #define:

```
int mat1[M][N];
int mat2[N][M];

int sum_element(int i, int j) {
    return mat1[i][j] + mat2[j][i];
}
```

In compiling this program, GCC generates the following assembly code:

```
  i at %ebp+8, j at %ebp+12
    movl    8(%ebp), %ecx
    movl    12(%ebp), %edx
    leal    0(,%ecx,8), %eax
    subl    %ecx, %eax
    addl    %edx, %eax
    leal    (%edx,%edx,4), %edx
    addl    %ecx, %edx
    movl    mat1(,%eax,4), %eax
    addl    mat2(,%edx,4), %eax
```

Use your reverse engineering skills to determine the values of M and N based on this assembly code.

Solution to Problem 3.37 (page 236)

This problem requires you to work through the scaling operations to determine the address computations, and to apply Equation 3.1 for row-major indexing. The first step is to annotate the assembly code to determine how the address references are computed:

```
movl    8(%ebp), %ecx           Get i
movl    12(%ebp), %edx          Get j
leal    0(,%ecx,8), %eax        8*i
subl    %ecx, %eax              8*i-i = 7*i
addl    %edx, %eax              7*i+j
leal    (%edx,%edx,4), %edx     5*j
addl    %ecx, %edx              5*j+i
movl    mat1(,%eax,4), %eax     mat1[7*i+j]
addl    mat2(,%edx,4), %eax     mat2[5*j+i]
```

We can see that the reference to matrix `mat1` is at byte offset $4(7i + j)$, while the reference to matrix `mat2` is at byte offset $4(5j + i)$. From this, we can determine that `mat1` has 7 columns, while `mat2` has 5, giving $M = 5$ and $N = 7$.

Structs

```
struct s {
  char c1;
  int i;
  char c2;
  int j;
};
```

- What's the problem with this struct?

Structs

```
struct s {
  char c1;
  int i;
  char c2;
  int j;
};
```

- Say an instance of the struct begins at 0x10. Then c1 is at address 0x10. However, 'i' cannot be at address 0x11 (it needs to be 4-aligned). As a result, we need 3 bytes of padding.

Structs

- This is a waste of space! There will be 3 bytes of padding after c1 and 3 bytes of padding after c2, meaning that this struct will take up 16 bytes when really it only needs 10.

Structs

- Two common struct ordering guidelines (which could be at odds):
  1. Place the most commonly used data type first.
  2. Place the elements in descending order of size (ie largest first)
- Why?

Structs

- 1.
- Memory references are expensive (ex. (%eax))... but memory references with an offset are more expensive (ex. 8(%eax))
- Chances are, you'll be referring to the struct by a pointer to the beginning of the struct, which means that dereferencing the pointer without an offset will point to the first element.

Structs

- 2.
- If the elements with larger sizes are first, that means there will be less of a need for padding.
- For example, consider struct s, except with the first two elements swapped:

```
struct s {
  int i;
  char c1;
  char c2;
  int j;
};
```

Structs

- 2.
- ```
  struct s {
    int i;
    char c1;
    char c2;
    int j;
  };
  ```
- Now, we need 2 bytes of padding between c2 and j for a total of 12 bytes.
  ```

Structs

- Because each internal element must follow their own alignment rules, the alignment of the struct must be equal to the strictest of the elements within a struct.
- But wait...

Structs

- Consider:

```
struct s {
  char  c;
  int i;
};
```

- Because int i is aligned by 4, instances of struct s must be aligned by 4.
- There must also be 3 bytes of padding between c and I, meaning a total size of 8.

Structs

- Thus, a possible placement of (struct s s1) where s1.c = 0xFF and s1.i = 0x33221100 is the following:

Address:	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
Value:	0XFF	0xXX	0xXX	0xXX	0x00	0x11	0x22	0x33

- Where s begins at 0x10.
- This is how we meet the alignment requirements of each individual item

Unions

- Like structs except all of the values begin at the same address.
- union s {
-   short s;
-   char c;
- };
- This means that in a union that contains several values, only one of them is likely to be meaningful and assigning one term a value will trample other terms.

Unions

- union s {
-   short s;
-   char c;
- };
- union s foo;
- Say foo begins at 0x10.
- foo.s will be located in addresses 0x10 and 0x11
- foo.c will be located in address 0x10.

Unions

- union s {
-   short s;
-   char c;
- };
- union s foo;
- foo.s = 0xFFFF;
- foo.c = 0;
- printf(“%hx\n”, foo.s) => FF00

gdb - Debugger

(gdb) break <function_name>

(gdb) run (gdb)

(gdb) stepi

(gdb) stepn

(gdb) info registers

(gdb) disassemble

All variants of “print”

Cheat Sheet – gdbnotes.pdf

Example – gdb.pdf