Systems I

Machine-Level Programming VIII: Structured Data

Topics
- Alignment
- Unions
- Buffer bounds checking and overflow
Alignment

Aligned Data
- Primitive data type requires K bytes
- Address must be multiple of K
- Required on some machines; advised on IA32
  - treated differently by Linux and Windows!

Motivation for Aligning Data
- Memory accessed by (aligned) double or quad-words
  - Inefficient to load or store datum that spans quad word boundaries
  - Virtual memory very tricky when datum spans 2 pages

Compiler
- Inserts gaps in structure to ensure correct alignment of fields
Specific Cases of Alignment

Size of Primitive Data Type:

- **1 byte** (e.g., char)
  - no restrictions on address
- **2 bytes** (e.g., short)
  - lowest 1 bit of address must be 0₂
- **4 bytes** (e.g., int, float, char *, etc.)
  - lowest 2 bits of address must be 00₂
- **8 bytes** (e.g., double)
  - Windows (and most other OS’s & instruction sets):
    - lowest 3 bits of address must be 000₂
  - Linux:
    - lowest 2 bits of address must be 00₂
    - i.e., treated the same as a 4-byte primitive data type
- **12 bytes** (long double)
  - Linux:
    - lowest 2 bits of address must be 00₂
    - i.e., treated the same as a 4-byte primitive data type
Satisfying Alignment with Structures

Offsets Within Structure
- Must satisfy element’s alignment requirement

Overall Structure Placement
- Each structure has alignment requirement K
  - Largest alignment of any element
- Initial address & structure length must be multiples of K

Example (under Windows):
- K = 8, due to double element

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Linux vs. Windows

Windows (including Cygwin):
- $K = 8$, due to double element

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

Linux:
- $K = 4$; double treated like a 4-byte data type
Overall Alignment Requirement

```c
struct S2 {
    double x;
    int i[2];
    char c;
} *p;
```

`p` must be multiple of:
- 8 for Windows
- 4 for Linux

```c
struct S3 {
    float x[2];
    int i[2];
    char c;
} *p;
```

`p` must be multiple of 4 (in either OS)
Ordering Elements Within Structure

```c
struct S4 {
    char c1;
    double v;
    char c2;
    int i;
} *p;
```

10 bytes wasted space in Windows

```
struct S5 {
    double v;
    char c1;
    char c2;
    int i;
} *p;
```

2 bytes wasted space
Arrays of Structures

Principle

- Allocated by repeating allocation for array type
- In general, may nest arrays & structures to arbitrary depth

```
struct S6 {
    short i;
    float v;
    short j;
} a[10];
```
Accessing Element within Array

- Compute offset to start of structure
  - Compute 12\(i\) as 4\((i+2i)\)
- Access element according to its offset within structure
  - Offset by 8
  - Assembler gives displacement as \(a + 8\)
    » Linker must set actual value

```c
short get_j(int idx)
{
    return a[idx].j;
}
```

```assembly
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(%eax,4),%eax
```

```
<table>
<thead>
<tr>
<th>a[0]</th>
<th>⋯ ⋯</th>
<th>a[i]</th>
<th>⋯ ⋯</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+0</td>
<td></td>
<td>a+12i</td>
<td></td>
</tr>
<tr>
<td>a[i].i</td>
<td>a[i].v</td>
<td>a[i].j</td>
<td></td>
</tr>
<tr>
<td>a+12i</td>
<td></td>
<td>a+12i+8</td>
<td></td>
</tr>
</tbody>
</table>
```
Satisfying Alignment within Structure

Achieving Alignment

- Starting address of structure array must be multiple of worst-case alignment for any element
  - a must be multiple of 4
- Offset of element within structure must be multiple of element’s alignment requirement
  - v’s offset of 4 is a multiple of 4
- Overall size of structure must be multiple of worst-case alignment for any element
  - Structure padded with unused space to be 12 bytes

```c
struct S6 {
    short i;
    float v;
    short j;
} a[10];
```
Union Allocation

Principles

- Overlay union elements
- Allocate according to largest element
- Can only use one field at a time

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *sp;

union U1 {
    char c;
    int i[2];
    double v;
} *up;
```

(Windows alignment)
typedef union {
    float f;
    unsigned u;
} bit_float_t;

float bit2float(unsigned u) {
    bit_float_t arg;
    arg.u = u;
    return arg.f;
}

unsigned float2bit(float f) {
    bit_float_t arg;
    arg.f = f;
    return arg.u;
}

- Get direct access to bit representation of float
- `bit2float` generates float with given bit pattern
  - NOT the same as `(float) u`
- `float2bit` generates bit pattern from float
  - NOT the same as `(unsigned) f`
Byte Ordering Revisited

Idea
- Short/long/quad words stored in memory as 2/4/8 consecutive bytes
- Which is most (least) significant?
- Can cause problems when exchanging binary data between machines

Big Endian
- Most significant byte has lowest address
- PowerPC, Sparc

Little Endian
- Least significant byte has lowest address
- Intel x86, Alpha
union {
    unsigned char c[8];
    unsigned short s[4];
    unsigned int i[2];
    unsigned long l[1];
} dw;

   i[0]    i[1]
   l[0]
int j;
for (j = 0; j < 8; j++)
dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
   dw.c[0], dw.c[1], dw.c[2], dw.c[3],
   dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 ==
[0x%x,0x%x,0x%x,0x%x]\n",
   dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
   dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
   dw.l[0]);
Byte Ordering on x86

Little Endian

<table>
<thead>
<tr>
<th>Characters</th>
<th>0-7</th>
<th>(0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shorts</td>
<td>0-3</td>
<td>(0xf1f0,0xf3f2,0xf5f4,0xf7f6)</td>
</tr>
<tr>
<td>Ints</td>
<td>0-1</td>
<td>(0xf3f2f1f0,0xf7f6f5f4)</td>
</tr>
<tr>
<td>Long</td>
<td>0</td>
<td>(0xf3f2f1f0)</td>
</tr>
</tbody>
</table>

Output on Pentium:

<table>
<thead>
<tr>
<th>Characters</th>
<th>0-7</th>
<th>(0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shorts</td>
<td>0-3</td>
<td>(0xf1f0,0xf3f2,0xf5f4,0xf7f6)</td>
</tr>
<tr>
<td>Ints</td>
<td>0-1</td>
<td>(0xf3f2f1f0,0xf7f6f5f4)</td>
</tr>
<tr>
<td>Long</td>
<td>0</td>
<td>(0xf3f2f1f0)</td>
</tr>
</tbody>
</table>
Byte Ordering on Sun

Big Endian

Output on Sun:

Characters 0–7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0–3 == [0xf0f1, 0xf2f3, 0xf4f5, 0xf6f7]
Ints 0–1 == [0xf0f1f2f3, 0xf4f5f6f7]
Long 0 == [0xf0f1f2f3]
Byte Ordering on Alpha

Little Endian

<table>
<thead>
<tr>
<th>f0</th>
<th>f1</th>
<th>f2</th>
<th>f3</th>
<th>f4</th>
<th>f5</th>
<th>f6</th>
<th>f7</th>
</tr>
</thead>
</table>

|------|------|------|------|

<table>
<thead>
<tr>
<th>i[0]</th>
<th>i[1]</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>l[0]</th>
</tr>
</thead>
</table>

Output on Alpha:

Characters 0–7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0–3 == [0xf1f0, 0xf3f2, 0xf5f4, 0xf7f6]
Ints 0–1 == [0xf3f2f1f0, 0xf7f6f5f4]
Long 0 == [0xf7f6f5f4f3f2f1f0]
Summary

Arrays in C

- Contiguous allocation of memory
- Pointer to first element
- No bounds checking

Compiler Optimizations

- Compiler often turns array code into pointer code (zd2int)
- Uses addressing modes to scale array indices
- Lots of tricks to improve array indexing in loops

Structures

- Allocate bytes in order declared
- Pad in middle and at end to satisfy alignment

Unions

- Overlay declarations
- Way to circumvent type system
Extra slides
Dynamic Nested Arrays

Strength
- Can create matrix of arbitrary size

Programming
- Must do index computation explicitly

Performance
- Accessing single element costly
- Must do multiplication

```c
int * new_var_matrix(int n)
{
    return (int *)
    calloc(sizeof(int), n*n);
}
```

```c
int var_ele
    (int *a, int i,
     int j, int n)
{
    return a[i*n+j];
}
```

```assembly
movl 12(%ebp),%eax  # i
movl 8(%ebp),%edx   # a
imull 20(%ebp),%eax # n*i
addl 16(%ebp),%eax  # n*i+j
movl (%edx,%eax,4),%eax # Mem[a+4*(i*n+j)]
```
Dynamic Array Multiplication

Without Optimizations

- **Multiplies**
  - 2 for subscripts
  - 1 for data

- **Adds**
  - 4 for array indexing
  - 1 for loop index
  - 1 for data

```c
/* Compute element i,k of variable matrix product */
int var_prod_ele
    (int *a, int *b,
     int i, int k, int n)
{
    int j;
    int result = 0;
    for (j = 0; j < n; j++)
        result +=
            a[i*n+j] * b[j*n+k];
    return result;
}
```

Row-wise A

Column-wise B

(A, *) (i, *), (*, k)
Optimizing Dynamic Array Mult.

**Optimizations**
- Performed when set optimization level to -O2

**Code Motion**
- Expression i*n can be computed outside loop

**Strength Reduction**
- Incrementing j has effect of incrementing j*n+k by n

**Performance**
- Compiler can optimize regular access patterns

```c
{  
    int j;
    int result = 0;
    int iTn = i*n;
    int jTnPk = k;
    for (j = 0; j < n; j++) {  
        result +=
            a[iTn+j] * b[jTnPk];
        jTnPk += n;
    }
    return result;
}
```