CS429: Computer Organization and Architecture

Instruction Set Architecture VI

Warren Hunt, Jr. and Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: October 14, 2014 at 13:35

Alignment

Aligned Data

- Primitive data type requires K bytes.
- Address "should" be a multiple of K.
- This is required on some machines and advised on IA32.
- Treated differently by Linux and Windows.

Motivation for Aligning Data

- Memory accessed by (aligned) double or quad-words.
- It's inefficient to load or store a datum that spans quad word boundaries.
- Virtual memory gets 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 bit of address must be 0_2

4 bytes (e.g., int, float, char *, etc)

- lowest 2 bits of address must be 00_2

8 bytes (e.g., double)

- On Windows and most other OSs: lowest 3 bits of address must be 000_2
- On Linux: lowest 2 bits of address must be 00_2; treated as a 4-byte primitive data type.

12 bytes (e.g., long double)

- On Linux: lowest 2 bits of address must be 00_2; treated as a 4-byte primitive data type.

Satisfying Alignment with Structures

Offsets within Structure

- Must satisfy element's alignment requirements.

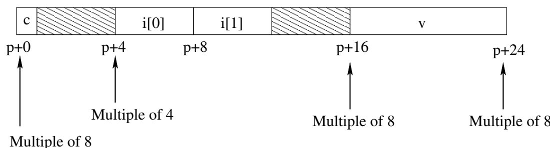
Overall Structure Placement

- Each structure has alignment requirement K.
- K is the largest alignment of any element.
- Initial address and structure length must be multiples of K.

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

Example (under Windows):

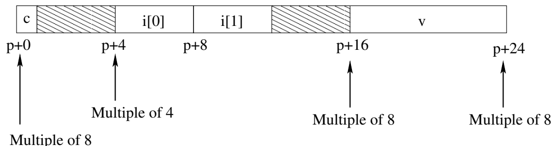
- K = 8, due to double element.



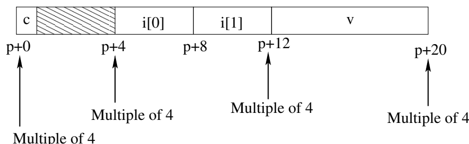
Linux vs. Windows

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

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



Linux:
K = 4; the double is treated like a 4-byte data type.

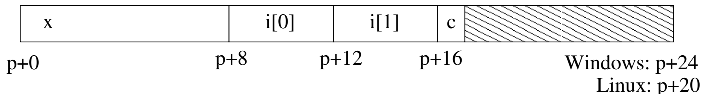


Overall Alignment Requirement

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

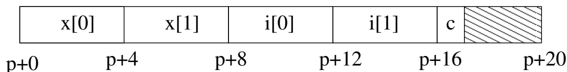
p must be a multiple of:

- 8 for Windows
- 4 for Linux




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

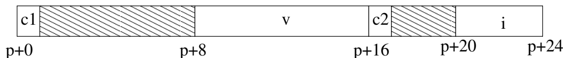
p must be a multiple of 4 (in either OS).



Ordering Elements within Structures

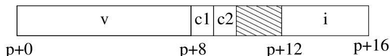
```
struct S4 {
   char c1;
   double v;
   char c2;
   int i;
} *p;
```

10 bytes of wasted space in Windows




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

2 bytes wasted space.

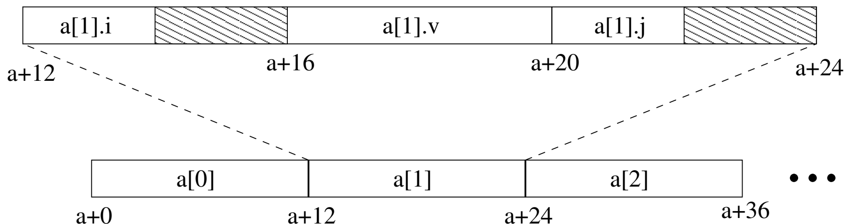


Arrays of Structures

Principle

- Allocated by repeating allocation for the element type.
- In general, you can nest arrays and structures to arbitrary depth.

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

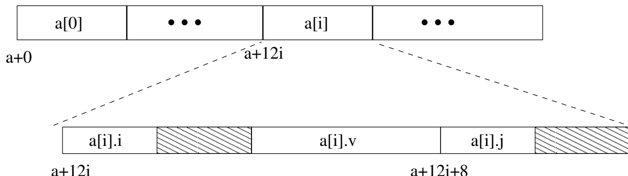


Accessing Elements within Array

- Compute offset to start of structure.
- Compute $12i$ as $4 * (i + 2i)$.
- Access element according to its offset within the structure.
- Assembler gives displacement as a+8; linker must set the actual value.

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

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

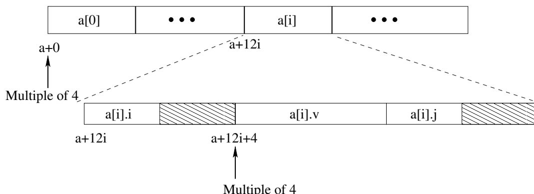
```
# %eax = idx
leal   (%eax,%eax,2),%eax
movswl  a+8(,%eax,4),%eax
```



Achieving Alignment

- Starting address of structure array must be a multiple of worst-case alignment for any element.
- Here a must be a multiple of 4.
- Offset of element within structure must be multiple of element's alignment requirement.
- v's offset 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 12 bytes.

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

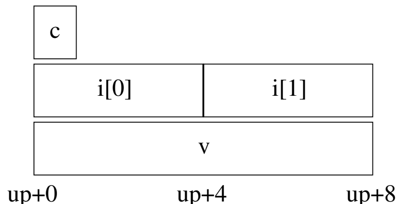


Union Allocation

Principles

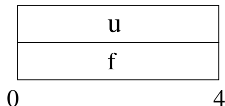
- Overlay union elements.
- Allocate according to the largest element.
- Can only use one field at a time.

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



Using Union to Access Bit Patterns

```
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 t)
{
    bit_float_t arg;
    arg.f = f;
    return arg.u;
}
```

- Get direct representation to bit representation of float.
- `bit2float` generates float with given bit pattern.
- Note this is not the same as `(float) u`.
- `float2bit` generates bit pattern from float.
- Note this is not the same as `(unsigned) f`.

Byte Order Revisited

Idea

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

Big Endian

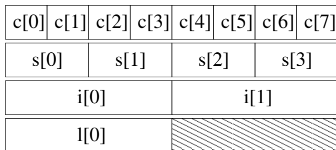
- Most significant byte has lowest address.
- PowerPC, Sparc

Little Endian

- Least significant byte has lowest address.
- Intel x86, Alpha

Byte Ordering Example

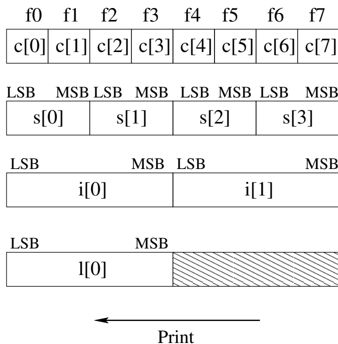
```
union {
    unsigned char c[8];
    unsigned short s[4];
    unsigned int i[2];
    unsigned long l[1];
} dw;
```




```
int j;
for (j = 0; j < 8; j++)
   dw.c[j] = 0xf0 + j;
printf("Chars 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 the x86

Little Endian

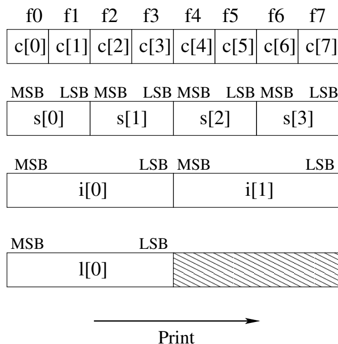


Output on Pentium:

```
Chars  0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts 0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints   0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long     0 == [0xf3f2f1f0]
```

Byte Ordering on Sun

Big Endian



Output on Sun:

```
Chars  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]
```

Summary

Arrays in C

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

Compiler Optimizations

- Compiler often turns array code into pionter code.
- 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.

Dynamic Nested Arrays

Strength: Create array of arbitrary size.

Programming: Must do index computation explicitly.

Performance:

- Accessing a single element is costly.
- Must do multiplication.

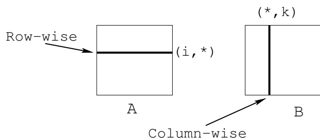
```
int *new_var_matrix(int n)
{
  return (int *)
    calloc(sizeof(int), n*n);
}

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

```
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*(n*i+j)]
```

Dynamic Array Multiplication

Without optimization:

- Multiplies: 2 for subscripts, 1 for data
- Adds: 4 for array indexing, 1 for loop index, 1 for data




```
/* Compute element i,i 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
}
```

Optimizing Dynamic Array Multiplication

Optimizations

- Performed when set optimization level to -O2

Code Motion

- Expression `i*n` can be performed outside loop.

Strength Reduction

- Incrementing j has the effect of incrementing `j*n+k` by n.

Performance

- Compiler can optimize regular access patterns.

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

```
{
    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;
}
```