Structured Data II
Heterogenous Data
Feb. 15, 2000

Topics
- Structure Allocation
- Alignment
- Unions
- Byte Ordering
- Byte Operations
- IA32/Linux Memory Organization

Basic Data Types

Integral
- Stored & operated on in general registers
- Signed vs. unsigned depends on instructions used
  Intel | GAS | Bytes | C
  byte | b   | 1    | [unsigned] char
  word | w   | 2    | [unsigned] short
  double word | 4  | [unsigned] int, char *
  quad word   | 8  |

Floating Point
- Stored & operated on in floating point registers
  Intel | GAS | Bytes | C
  Single | s | 4 | float
  Double | d | 8 | double
  Extended | 10 | -- |

Structures

Concept
- Contiguously-allocated region of memory
- Refer to members within structure by names
- Members may be of different types

```
struct rec {
    int i;
    int a[3];
    int *p;
};
```

Accessing Structure Member

```
void set_i(struct rec *r, int val) {
    r->i = val;
}
```

Generating Pointer to Structure Member

```
int *find_a(struct rec *r, int idx) {
    return &r->a[idx];
}
```

Generating Pointer to Array Element

```
int *find_a(struct rec *r, int idx) {
    return &r->a[idx];
}
```

Assembly

```
# %eax = val
# %edx = r
movl %eax, (%edx) # Mem[r] = val
```
Structure Referencing (Cont.)

C Code

```c
struct rec {
    int i;
    int a[3];
    int *p;
};
```

```c
void set_p(struct rec *r)
{
    r->p = &r->a[r->i];
}
```

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 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 0
- 4 bytes (e.g., int, float, char *, etc.)
  - lowest 2 bits of address must be 00
- 8 bytes (e.g., double)
  - Windows (and most other OS's & instruction sets):
    - lowest 3 bits of address must be 000
  - Linux:
    - lowest 2 bits of address must be 00
    - 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

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
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; double treated like a 4-byte data type

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

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

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

Effect of Overall Alignment Requirement

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

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

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

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

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

Ordering Elements Within Structure

10 bytes wasted space in Windows

c1 v c2 i
p+0 p+8 p+16 p+20 p+24

2 bytes wasted space

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

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];

struct S6 {
    short i;
    float v;
    short j;
} a[10];
Accessing Element within Array
- Compute offset to start of structure
  - Compute \(12^i\) as \(4^i\times2\)
- Access element according to its offset within structure
  - Offset by 8
  - Assembler gives displacement as \(a + 8\)
  > Linker must set actual value

\[
\text{short get}_j(int \ idx) \\
\text{\{} \\
\text{\quad return a[\idx].j;} \\
\text{\}\}
\]

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

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

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

Implementing “Tagged” Union
- Structure can hold 3 kinds of data
- Only one form at any given time
- Identify particular kind with flag type

\[
\text{typedef enum \{ CHAR, INT, DBL \} utype;} \\
\text{typedef struct \{} \\
\quad utype type; \\
\quad union \{} \\
\quad \quad \text{char c;} \\
\quad \quad \text{int i[2];} \\
\quad \quad \text{double v;} \\
\quad \text{\} e; } \\
\quad \text{\} store_sle, store_ptr;} \\
\text{store_sle k;} \\
\]

Page 4
Using “Tagged” Union

```c
store_ele k1;
  k1.type = CHAR;
  k1.e.c = 'a';

store_ele k2;
  k2.type = INT;
  k2.e.i[0] = 17;
  k2.e.i[1] = 47;

store_ele k3;
  k3.type = DBL;
  k1.e.v = 3.14159265358979323846;
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>17</th>
<th>47</th>
</tr>
</thead>
</table>

Using Union to Access Bit Patterns

```c
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;
}
```

Using Union to Access Bit Patterns

- 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

**Idea**
- Long/quad words stored in memory as 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
  - IBM 360/370, Motorola 68K, Sparc

**Little Endian**
- Least significant byte has lowest address
  - Intel x86, Digital VAX

Byte Ordering Example

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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>i[0]</td>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Byte Ordering Example (Cont).

```c
int j;
for (j = 0; j < 8; j++)
dw.c[j] = 0x00 + j;
printf("Characters 0-7 ==\n[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 ==\n[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>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>

LSB: `c[0]` to `c[7]`

MSB: `c[7]` to `c[0]`

**Output on Pentium:**

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 == `[f3f2f1f0]`

### Byte Ordering on Sun

**Big Endian**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>

MSB: `c[0]` to `c[7]`

LSB: `c[7]` to `c[0]`

**Output on Sun:**

Characters 0-7 == `[0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]`

Shorts 0-3 == `[0xf0f1f2f3,0xf4f5f6f7]`

Ints 0-1 == `[0xf0f1f2f3,0xf4f5f6f7]`

Long 0 == `[0xf0f1f2f3]`

### Byte Ordering on Alpha

**Little Endian**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>

LSB: `c[0]` to `c[7]`

MSB: `c[7]` to `c[0]`

**Output on Alpha:**

Characters 0-7 == `[0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]`

Shorts 0-3 == `[0xf0f1f2f3,0xf4f5f6f7]`

Ints 0-1 == `[0xf0f1f2f3,0xf4f5f6f7]`

Long 0 == `[0xf0f1f2f3]`
Byte-Level Operations

IA32 Support
- Arithmetic and data movement operations have byte-level version
  - movb, addb, testb, etc.
- Some registers partially byte-addressable
- Can perform single byte memory references

Compiler
- Parameters and return values of type char passed as int's
- Use movsbl to sign-extend byte to int

Byte-Level Operation Example
- Compute xor of characters in string

```c
char string_xor(char *s)
{
    char result = 0;
    char c;
    do {
        c = *s++;
        result ^= c;
        testb %al, %al
        jne L2
    } while (c);
    return result;
}
```

Linux Memory Layout

- Stack
  - Runtime stack (8MB limit)
- Heap
  - Dynamically allocated storage
  - When call malloc, calloc, new
- DLLs
  - Dynamically Linked Libraries
  - Library routines (e.g., printf, malloc)
  - Linked into object code when first executed
- Data
  - Statically allocated data
  - E.g., arrays & strings declared in code
- Text
  - Executable machine instructions
  - Read-only

Linux Memory Allocation

Initially
- Stack
- Linked
- Some
- More

Red Hat
- v. 5.2
- 120MB memory
Memory Allocation Example

```c
char big_array[1<<24]; /* 16 MB */
char huge_array[1<<28]; /* 256 MB */
int beyond;
char *p1, *p2, *p3, *p4;
int useless() { return 0; }

int main()
{
    p1 = malloc(1 << 28); /* 256 MB */
p2 = malloc(1 << 8); /* 256 B */
p3 = malloc(1 << 28); /* 256 MB */
p4 = malloc(1 << 8); /* 256 B */
    /* Some print statements ... */
}
```

Dynamic Linking Example

```c
(gdb) print malloc
$1 = {<text variable, no debug info>}
0x8048454 <malloc>
(gdb) run
Program exited normally.
(gdb) print malloc
$2 = {void *(unsigned int)}
0x40006240 <malloc>
```

Initially
- Code in text segment that invokes dynamic linker
- Address 0x8048454 should be read 0x08048454

Final
- Code in DLL region

Breakpointing Example

```c
(gdb) break main
(gdb) run
Breakpoint 1, 0x804856f in main ()
(gdb) print $esp
$3 = (void *) 0xbffffc78
```

Main
- Address 0x804856f should be read 0x0804856f

Stack
- Address 0xbffffc78

Example Addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$esp</td>
<td>0xbffffc78</td>
</tr>
<tr>
<td>p3</td>
<td>0x500b5008</td>
</tr>
<tr>
<td>p1</td>
<td>0x400b4008</td>
</tr>
<tr>
<td>Final malloc</td>
<td>0x40006240</td>
</tr>
<tr>
<td>p4</td>
<td>0x1904a640</td>
</tr>
<tr>
<td>p2</td>
<td>0x1904a538</td>
</tr>
<tr>
<td>beyond</td>
<td>0x1904a524</td>
</tr>
<tr>
<td>big_array</td>
<td>0x1804a520</td>
</tr>
<tr>
<td>huge_array</td>
<td>0x0804a510</td>
</tr>
<tr>
<td>main()</td>
<td>0x0804856f</td>
</tr>
<tr>
<td>useless()</td>
<td>0x08048560</td>
</tr>
<tr>
<td>Initial malloc</td>
<td>0x08048454</td>
</tr>
</tbody>
</table>