Structures and Alignment

Unaligned Data

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>i[0]</td>
<td>i[1]</td>
<td>v</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>p+1</td>
<td>p+5</td>
<td>p+9</td>
<td>p+17</td>
</tr>
</tbody>
</table>

Aligned Data

- Primitive data type requires $K$ bytes
- Address must be a multiple of $K$

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>extra bytes</td>
<td>i[0]</td>
<td>i[1]</td>
<td>extra bytes</td>
</tr>
<tr>
<td>p+0</td>
<td>p+4</td>
<td>p+8</td>
<td>p+16</td>
<td>p+24</td>
</tr>
</tbody>
</table>

Alignment Principles

Aligned Data
- Primitive data type requires $K$ bytes
- Address must be a multiple of $K$
- Required on some machines; advised on x86-64

Motivation for Aligning Data
- Memory accessed by (aligned) chunks of 4 to 8 bytes (system dependent)
- It’s inefficient to load or store datum that spans quad word boundaries
- Virtual memory is trickier when datum spans 2 pages

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

Specific Cases of Alignment (x86-64)

1 byte: char, ...
- no restrictions on address

2 bytes: short, ...
- lowest 1 bit of address must be 0₂

4 bytes: int, float, ...
- lowest 2 bits of address must be 00₂

8 bytes: double, long, char *, ...
- lowest 3 bits of address must be 000₂

16 bytes: long double (GCC on Linux)
- lowest 4 bits of address must be 0000₂
**Satisfying Alignment with Structures**

**Within structure:**
- Must satisfy each element’s alignment requirement

**Overall structure placement:**
- Each structure has alignment requirement $K$, where $K$ is the largest alignment of any element
- Initial address and structure length must be multiples of $K$

**Example:** $K = 8$, due to `double` element

<table>
<thead>
<tr>
<th></th>
<th><code>c</code></th>
<th><code>i[0]</code></th>
<th><code>i[1]</code></th>
<th><code>v</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr</td>
<td>p+0</td>
<td>p+4</td>
<td>p+8</td>
<td>p+16</td>
</tr>
<tr>
<td>Align</td>
<td>Multiple of 4</td>
<td>Multiple of 8</td>
<td>Multiple of 8</td>
<td>Multiple of 8</td>
</tr>
</tbody>
</table>

**Meeting Overall Alignment Requirement**
- For largest alignment requirement $K$
- Overall structure must be multiple of $K$

**Arrays of Structures**
- Overall structure length multiple of $K$
- Satisfy alignment requirement for every element

**Accessing Array Elements**
- **Compute array offset $12*idx$**
  - $\text{sizeof(S3)}$, including alignment spacers
- **Element $j$ is at offset 8 within structure**
- **Assembler gives offset $a+8$**
  - Resolved during linking
Accessing Array Elements

<table>
<thead>
<tr>
<th>a[0]</th>
<th>⋯</th>
<th>a[idx]</th>
<th>⋯</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+0</td>
<td>a+12</td>
<td>a+12*idx</td>
<td>⋯</td>
</tr>
</tbody>
</table>

\[ i \quad 2 \text{ bytes} \quad \text{v} \quad j \quad 2 \text{ bytes} \]

\[ a+12*\text{idx} \quad a+12*\text{idx}+8 \]

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

\# %rdi holds idx
\text{lea} (\%rdi,\%rdi,2),\%rax \ # 3*idx
\text{movzwl} \ a+8(,\%rax,4), \%eax

Saving Space

Put large data types first!

\textbf{Instead of:}

\textbf{do this:}

\textbf{Effect (K = 4)}

<table>
<thead>
<tr>
<th>c</th>
<th>3 bytes</th>
<th>i</th>
<th>d</th>
<th>3 bytes</th>
</tr>
</thead>
</table>

| i | c | d | 2 bytes |

Union Allocation

\textbf{Principles}

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

\textbf{Union U1}

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

Using Union to Access Bit Patterns

\textbf{typedef union} \ 
\{ 
\text{float} \ \text{bit}_2\text{float} (\text{unsigned} \ u) \ 
\{ \ 
\text{bit}_2\text{float}_t \ \text{arg}; \ 
\text{arg}.u = u; \ 
\text{return} \ \text{arg}.f; \ 
\} \ 
\} \ \text{bit}_2\text{float}_t;

\textbf{unsigned} \ \text{float}_2\text{bit} (\text{float} \ t) \ 
\{ 
\text{bit}_2\text{float}_t \ \text{arg}; \ 
\text{arg}.f = f; \ 
\text{return} \ \text{arg}.u; \ 
\}

- Get direct representation to bit representation of float.
- \text{bit}_2\text{float} \ \text{generates} \ \text{float} \ \text{with} \ \text{given} \ \text{bit} \ \text{pattern}.
- \text{Note: this \ is \ not \ the \ same} \ \text{as} \ (\text{float}) \ u.
- \text{float}_2\text{bit} \ \text{generates} \ \text{bit} \ \text{pattern} \ \text{from} \ \text{float}.
- \text{Note: this \ is \ not \ the \ same} \ \text{as} \ (\text{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

```c
union {
    unsigned char c[8];
    unsigned short s[4];
    unsigned int i[2];
    unsigned int 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("Int 0 == [0x%x] \n", dw.l[0]);
```

Byte Ordering on the x86

**Little Endian**

```
unsigned char c[8];
```

```
unsigned short s[4];
```

```
unsigned int i[2];
```

```
unsigned int l[1];
```

```
int i;
```

```c
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;
```

Print

Output on Pentium:

- **Chars** 0–7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
- **Shorts** 0–3 == [0xf0f1, 0xf0f2, 0xf0f3, 0xf0f4]
- **Ints** 0–1 == [0xf0f1f2f3, 0xf0f1f2f4]
- **Int** 0 == [0xf0f1f2f3]

Byte Ordering on Sun

**Big Endian**

```
unsigned char c[8];
```

```
unsigned short s[4];
```

```
unsigned int i[2];
```

```
unsigned int l[1];
```

```
int i;
```

```c
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;
```

Print

Output on Sun:

- **Chars** 0–7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
- **Shorts** 0–3 == [0xf0f1, 0xf0f2, 0xf0f3, 0xf0f4]
- **Ints** 0–1 == [0xf0f1f2f3, 0xf0f1f2f4]
- **Int** 0 == [0xf0f1f2f3]
Summary

**Arrays in C**
- Contiguous allocation of memory, row order.
- Pointer to first element.
- No bounds checking.

**Compiler Optimizations**
- Compiler often turns array code into pointer 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.