Alignment

Young W. Lim

2020-11-17 Tue

Outline

Based on

"Self-service Linux: Mastering the Art of Problem Determination", Mark Wilding
"Computer Architecture: A Programmer's Perspective", Bryant & O'Hallaron

Alignment (1)

- fundamental data types 2, 4, 8, bytes etc
- simple hardware design
- alignment restrictions
- the IA32 hardware works correctly
- but with performance degradation
- short data address : lsb is always 0
- int data address : 2 lsb's are always 00

Alignment (2)

- the IA32 hardware will work correctly regardless of the alignment of data
- Intel recommends that data be aligned to improve memory system performance
- Linux follows an alignment policy where 2-byte data type (e.g. `short`) must have an address that is a multiple of 2 while any larger data types (e.g. `int`, `int *`, `float`, `double`) must have an address that is a multiple of 4.
- Note that this requirement means that the least significant bit of the address of an object of type `short` must equal to 0
- Similary any object of type `int` or any pointer must be at an address having the low-order two bits equal to 0

Alignment (3)

- alignment is enforced by making sure that every data type is organized and allocatedin such a way that every object within the type satisfies its alignment restrctions
- the compiler places directives in the assembly code indicating the desired alignment for global data
- the assembly code declarartion of the jump table contains the following directive
  `.align 4`

Alignment (4)

- `.align 4`
- this ensures that the data following it
  (in this case the start of the jump table)
  will start with an address that is a multiple of 4
- since each table entry is 4 byte long
  the successive elements will obey the 4-byte alignment restriction

Alignment (5)

- library routines that allocate memory such as `malloc()` must be designed so that they return a pointer that satisfies the worst-case alignment restriction for the machine it is running on, typically 4 or 8

- for code involving structures, the compiler may need to insert gaps in the field allocation to ensure that each structure element satisfies its alignment requirement
- the structure then has some required alignment for its starting address

Alignment (6)

- consider the following structure declaration

```
struct S1 {
  int   i;
  char  c;
  int   j;
};
```

- suppose the compiler used the minimal 9-byte allocation
- then it would be impossible to satisfy the 4-byte alignment requirement for both fields `i` (offset 0) and `j` (offset 5)
- instead the compiler inserts a 3-byte gap

Alignment (7)

- as a result, `j` has offset 8, and the overall structure size is 12 bytes
- the compiler must ensure that any pointer `p` of the type `struct S1 *` satisfies a 4-byte alignment
- let pointer `p->i` (address x_p) and `p->j` (address $x_p + 4$) will satisfy their 4-byte alignmet requirement

Alignment (8)

- consider the following structure declaration

```
struct S2 {
  int   i;
  int   j;
  char  c;
};
```

- if we pack this structure into 9 bytes, we can still satisfy the alignment requirements for fields `i` and `j` by making sure that the starting address of the structure satisfies a 4-byte alignment requirement

Alignment (9)

- `struct S2 d[4];`
- with the 9-byte allocation, it is not possible
  to satisfy the alignment requirement for each element of `d`
  because these elements will have addresses
  x_d, $x_d + 9$, $x_d + 18$, $x_d + 27$
- instead the compiler will allocate 12 bytes for structure `S1`
  with the final 3 bytes being waste space

Alignment (10)

- `struct S1 d[4];`
- instead, the compiler will allocate 12 bytes
  for structure `S1`
  with the final 3 bytes being wated space
- that way the element of `d` will have addresses
  x_d, $x_d + 12$, $x_d + 24$, $x_d + 36$
- as long as x_d is a multiple of 4
  all of the alignment restrictions will be satisfied

Enforcing Alignment

- command line flag
  - -malign-double : 8-byte alignment
- assembler directive
  - .align 4

Structure Alignment (1)

```
struct S1 {
  int  i;
  char c;
  int  j;
};
```

- minimal 9-byte allocation

```
offset:
0x00 i
0x04 c
0x05 j
```

- 4-byte alignment
  - 3 bytes gap is inserted after c

```
offset:
0x00 i
0x04 c
0x08 j
```

Structure Alignment (2)

```
struct S1 {
  int  i;
  int  j;
  char c;
};

struct S2 A[2];
```

- 4-byte alignment
  - 3 bytes padding is inserted after c

```
offset:
0x00 A[0].i
0x04 A[0].j
0x08 A[0].c
0x0C A[1].i
0x10 A[1].j
0x14 A[1].c
```