数据的对齐存放

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

数据的对齐

Alignment: 要求数据的地址是相应的边界地址

- 目前机器字长为32位或64位，主存按一个传送单位（32/64/128位）进行存取，而按字节编址，例如：若传送单位为64位，则每次最多读写64位，即：第0~7字节同时读写，第8~15字节同时读写，……，以此类推。按边界对齐，可使读写数据位于8i~8i+7(i=0,1,2,...) 单元
- 指令系统支持对字节、半字、字及双字的运算
- 各种不同长度的数据存放时，有两种处理方式:
  - 按边界对齐（若一个字为32位）
    - 字地址：4的倍数(低两位为0)
    - 半字地址：2的倍数(低位为0)
    - 字节地址：任意
  - 不按边界对齐

    坏处：可能会增加访存次数！（学了存储器组织后会更明白！）

对齐(Alignment)

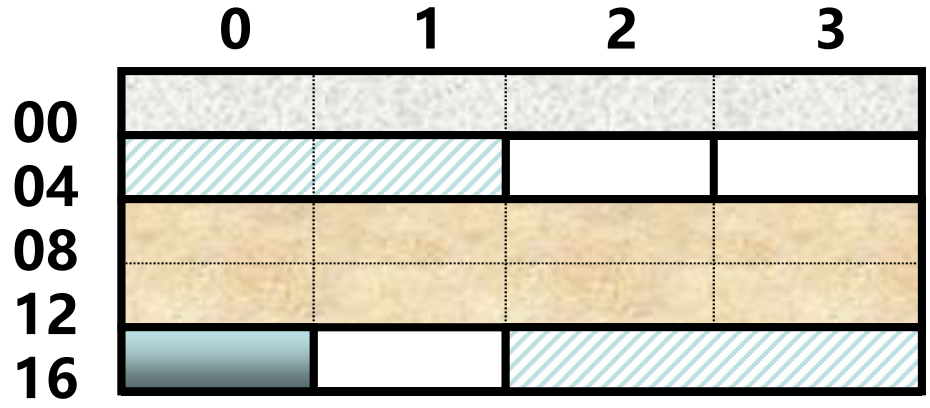
若1个字=32位，主存每次最多存取一个字，按字节编址，则每次**只能**读写某个字地址开始的4个单元中连续的1、2、3或4个字节

如：int i, short k, double x, char c, short j,……

按边界对齐

x：2个周期

j：1个周期



则：&i=0; &k=4; &x=8; &c=16; &j=18;……

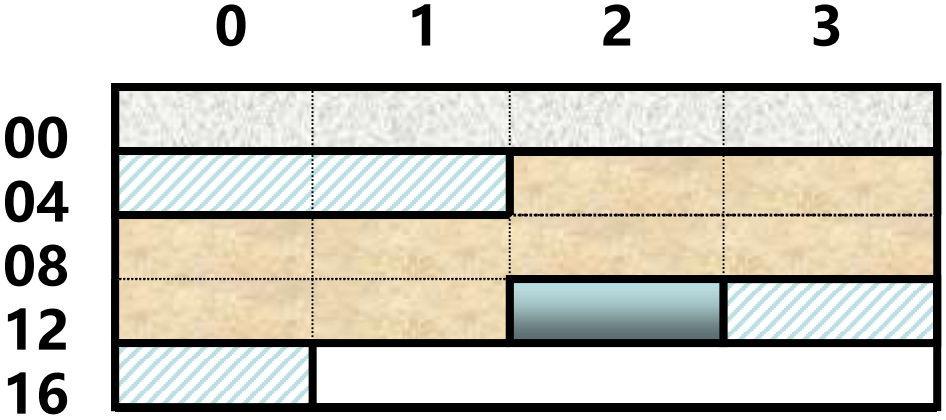
虽节省了空间，但增加了访存次数！

需要权衡，目前来看，浪费一点存储空间没有关系！

边界不对齐

x：3个周期

j：2个周期



则： &i=0; &k=4; &x=6; &c=14; &j=15;……

数据的对齐

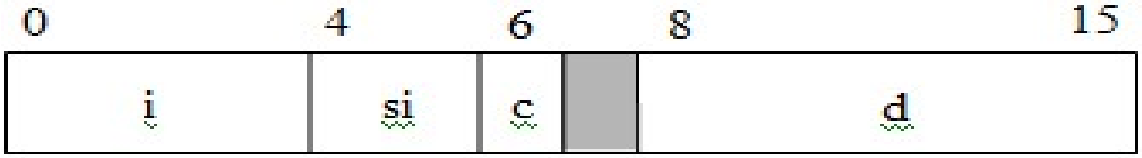
- 最简单的对齐策略是：按其数据长度进行对齐。例如，
  - Windows采用策略：int型地址是4的倍数，short型地址是2的倍数，double和long long型的是8的倍数，float型的是4的倍数，char不对齐
  - Linux采用更宽松策略：short型是2的倍数，其他类型如int、float、double和指针等都是4的倍数

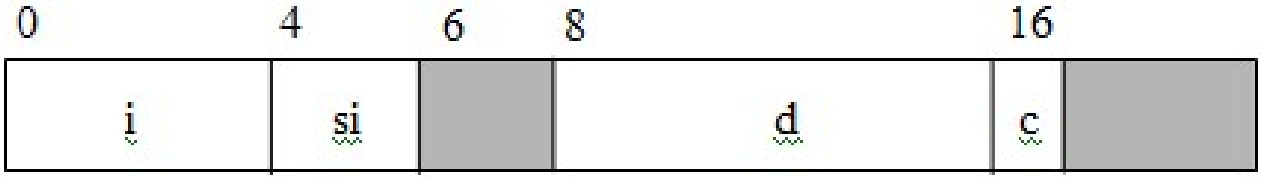
```
struct SDT {
    int       i;
    short     si;
    double    d;
    char      c;
} sa[10];
```

结构数组变量的最末可能需要插空，以使每个数组元素都按4字节边界对齐

```
struct SD {
    int       i;
    short     si;
    char      c;
    double    d;
};
```

结构变量首地址按4字节边界对齐





只要SD首址按4B边界对齐，所有字段都能按要求对齐

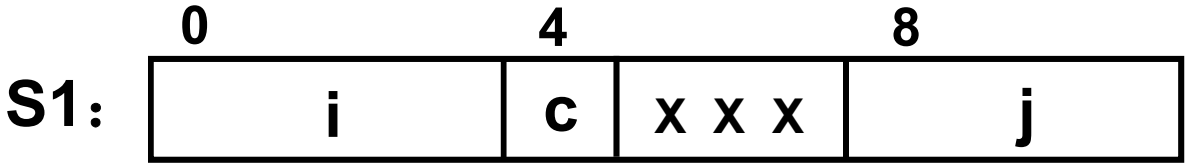
对齐(Alignment)举例

例如，考虑下列两个结构声明：

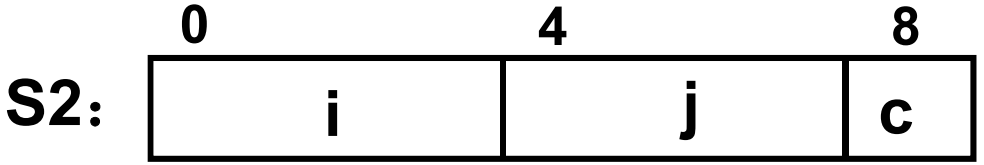
```
struct  S1 {
        int     i ;
        char    c ;
        int     j ;
} ;
```

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

在要求对齐的情况下，哪种结构声明更好？ S2比S1好

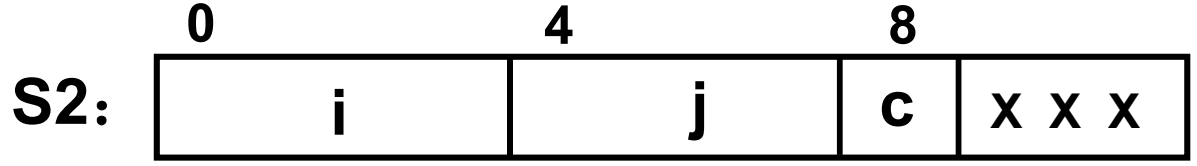


需要12个字节



只需要9个字节

对于“struct S2 d[4]”，只分配9个字节能否满足对齐要求？ 不能！



也需要12个字节

对齐方式的设定

#pragma pack(n)

- **为编译器指定结构体或类内部的成员变量的对齐方式。**
- **当自然边界（如int型按4字节、short型按2字节、float按4字节）比n大时，按n字节对齐。**
- **缺省或#pragma pack()，按自然边界对齐。**

__attribute__((aligned(m)))

- **为编译器指定一个结构体或类或联合体或一个单独的变量(对象)的对齐方式。**
- **按m字节对齐(m必须是2的幂次方)，且其占用空间大小也是m的整数倍，以保证在申请连续存储空间时各元素也按m字节对齐。**

__attribute__((packed))

- **不按边界对齐，称为紧凑方式。**

对齐方式的设定

```
#include<stdio.h>
#pragma pack(4)
typedef struct {
    uint32_t    f1;
    uint8_t     f2;
    uint8_t     f3;
    uint32_t    f4;
    uint64_t    f5;
}__attribute__((aligned(1024))) ts;
int main()
{
    printf("Struct size is: %d, aligned on 1024\n",sizeof(ts));
    printf("Allocate f1 on address: 0x%x\n",&(((ts*)0)->f1));
    printf("Allocate f2 on address: 0x%x\n",&(((ts*)0)->f2));
    printf("Allocate f3 on address: 0x%x\n",&(((ts*)0)->f3));
    printf("Allocate f4 on address: 0x%x\n",&(((ts*)0)->f4));
    printf("Allocate f5 on address: 0x%x\n",&(((ts*)0)->f5));
    return 0;
}
```

输出：

```
Struct size is: 1024, aligned on 1024
Allocate f1 on address: 0x0
Allocate f2 on address: 0x4
Allocate f3 on address: 0x5
Allocate f4 on address: 0x8
Allocate f5 on address: 0xc
```

```c
#include <stdio.h>
//#pragma pack(1)
struct test
{
    char x2;
    int x1;
    short x3;
    long long x4;
}__attribute__((packed));
struct test1
{
    char x2;
    int x1;
    short x3;
    long long x4;
};
struct test2
{
    char x2;
    int x1;
    short x3;
    long long  x4;
}__attribute__((aligned(8)));

void main()
{
    printf("size=%d\n",sizeof(struct test));
    printf("size=%d\n",sizeof(struct test1));
    printf("size=%d\n",sizeof(struct test2));
}
```

输出结果是什么？

```
size=15
size=20
size=24
```

```c
#include <stdio.h>
#pragma pack(1)
struct test
{
    char x2;
    int x1;
    short x3;
    long long x4;
}__attribute__((packed));
struct test1
{
    char x2;
    int x1;
    short x3;
    long long x4;
};
struct test2
{
    char x2;
    int x1;
    short x3;
    long long  x4;
}__attribute__((aligned(8)));
void main()
{
    printf("size=%d\n",sizeof(struct test));
    printf("size=%d\n",sizeof(struct test1));
    printf("size=%d\n",sizeof(struct test2));
}
```

如果设置了pragma pack(1)，结果又是什么？

```
size=15
size=15
size=16
```

```c
#include <stdio.h>
#pragma pack(2)
struct test
{
    char x2;
    int x1;
    short x3;
    long long x4;
}__attribute__((packed));
struct test1
{
    char x2;
    int x1;
    short x3;
    long long x4;
};
struct test2
{
    char x2;
    int x1;
    short x3;
    long long  x4;
}__attribute__((aligned(8)));
void main()
{
    printf("size=%d\n",sizeof(struct test));
    printf("size=%d\n",sizeof(struct test1));
    printf("size=%d\n",sizeof(struct test2));
}
```

如果设置了pragma pack(2)，结果又是什么？

```
size=15
size=16
size=16
```