Tony Bai

一个程序员的心路历程

再谈C语言位域

- 五月 21, 2013
- 5 条评论

我在日常工作中使用C语言中的位域(bit field)的场景甚少，原因大致有二：

* 一直从事于服务器后端应用的开发，现在的服务器的内存容量已经达到了数十G的水平，我们一般不需要为节省几个字节而使用内存布局更加紧凑的位域。
* 结构体中位域的实现是平台相关或Compiler相关的，移植性较差，我们不会贸然地给自己造“坑”的。

不过近期Linux技术内核社区（www.linux-kernel.cn) mail list中的一个问题让我觉得自己对bit field的理解还欠火候，于是乎我又花了些时间就着那个问题重新温习一遍bit field。

零、对bit field的通常认知

在C语言中，我们可以得到某个字节的内存地址，我们具备了操作任意内存字节的能力；在那个内存空间稀缺的年代，仅仅控制到字节级别还不足以满足C 程序员的胃口，为此C语言中又出现了bit级别内存的“有限操作能力” – 位域。这里所谓的“有限”指的是机器的最小粒度寻址单位是字节，我们无法像获得某个字节地址那样得到某个bit的地址，因此我们仅能通过字节的运算来设置 和获取某些bit的值。**在C语言中，尝试获得一个bit field的地址是非法操作**：

```
struct flag_t {
    int a : 1;
};

struct flag_t flg;
printf("%p\n", &flg.a);

error: cannot take address of bit-field ‘a’
```

以下是C语言中bit field的一般形式：

```
struct foo_t {
    unsigned int b1 : n1,
                 b2 : n2,
                 … …
                 bn : nk;
};
```

其中n1，n2，nk为对应位域所占据的bit数。

位域(bit field)的出现让我们可以**用变量名代表某些bit，并通过变量名直接获得和设置一些内存中bit的值**，而不是通 过晦涩难以理解的位操作来进行，例如：

```
struct foo_t {
    unsigned int a : 3,
                 b : 2,
                 c : 4;
};

struct foo_t f;
f.a = 3;
f.b = 1;
f.c = 12;
```

另外使用位域我们可以在展现和存储相同信息的同时，自定义**更加紧凑的内存布局**，节约内存的使用量。这使得bit field在嵌入式领域，在驱动程序领域得到广泛的应用，比如可以仅用两个字节就可以将tcpheader从dataoffset到fin的信息全部表示 和存储起来：

```
struct tcphdr {
    … …
    __u16   doff:4,
            res1:4,
            cwr:1,
            ece:1,
            urg:1,
            ack:1,
            psh:1,
            rst:1,
            syn:1,
            fin:1;
    … …
};
```

一、存储单元(storage unit)

C标准允许unsigned int/signed int/int类型的位域声明，C99中加入了_Bool类型的位域。但像Gcc这样的编译器自行加入了一些扩展，比如支持short、char等整型类 型的位域字段，使用其他类型声明位域将得到错误的结果，比如：

```
struct flag_t {
    char* a : 1;
};
 error: bit-field ‘a’ has invalid type
```

C编译器究竟是如何为bit field分配存储空间的呢？我们以Gcc编译器(Ubuntu 12.04.2 x86_64 Gcc 4.7.2)为例一起来探究一下。

我们先来看几个基本的bit field类型的例子：

```
struct bool_flag_t {
    _Bool a : 1,
          b : 1;
};

struct char_flag_t {
    unsigned char a : 2,
                  b : 3;
};

struct short_flag_t {
    unsigned short a : 2,
                   b : 3;
};

struct int_flag_t {
    int a : 2,
        b : 3;
};

int
main()
{
    printf("%ld\n", sizeof(struct bool_flag_t));
    printf("%ld\n", sizeof(struct char_flag_t));
    printf("%ld\n", sizeof(struct short_flag_t));
    printf("%ld\n", sizeof(struct int_flag_t));

    return 0;
}
```

编译执行后的输出结果为：

```
1
1
2
4
```

可以看出Gcc为不同类型的bit field分配了不同大小的基本内存空间。_Bool和char类型的基本存储空间为1个字节；short类型的基本存储空间为2个字节，int型的为4 个字节。这些空间的分配是基于结构体内部的bit field的size没有超出基本空间的界限为前提的。以short_flag_t为例：

```
struct short_flag_t {
    unsigned short a : 2,
                   b : 3;
};
```

a、b两个bit field总共才使用了5个bit的空间，所以Compiler只为short_flag_t分配一个基本存储空间就可以存储下这两个bit field。如果bit field的size变大，size总和超出基本存储空间的size时，编译器会如何做呢？我们还是看例子：

```
struct short_flag_t {
    unsigned short a : 7,
                   b : 10;
};
```

将short_flag_t中的两个bit字段的size增大后，我们得到的sizeof(struct short_flag_t)变成了4，显然Compiler发现一个基础存储空间已经无法存储下这两个bit field了，就又为short_flag_t多分配了一个基本存储空间。这里我们所说的基本存储空间就称为**“存储单元(storage unit)”**。它是Compiler在给bit field分配内存空间时的基本单位，并且这些分配给bit field的内存是以存储单元大小的整数倍递增的。但从上面来看，**不同类型bit field的存储单元大小是不同的。**

sizeof(struct short_flag_t)变成了4，那a和b有便会有至少两种内存布局方式：
* a、b紧邻
* b在下一个可存储下它的存储单元中分配内存

具体采用哪种方式，是Compiler相关的，这会影响到bit field的可移植性。我们来测试一下Gcc到底采用哪种方式：

```
void
dump_native_bits_storage_layout(unsigned char *p, int bytes_num)
{

    union flag_t {
        unsigned char c;
        struct base_flag_t {
            unsigned int p7:1,
                         p6:1,
                         p5:1,
                         p4:1,
                         p3:1,
                         p2:1,
                         p1:1,
                         p0:1;
        } base;
    } f;

    for (int i = 0; i < bytes_num; i++) {
        f.c = *(p + i);
```

```
        printf("%d%d%d%d %d%d%d%d ",
                        f.base.p7,
                        f.base.p6,
                        f.base.p5,
                        f.base.p4,
                        f.base.p3,
                        f.base.p2,
                        f.base.p1,
                        f.base.p0);
    }
    printf("\n");
}

struct short_flag_t {
    unsigned short a : 7,
                   b : 10;
};

 struct short_flag_t s;
 memset(&s, 0, sizeof(s));
 s.a = 113; /* 0111 0001 */
 s.b = 997; /* 0011 1110 0101 */

 dump_native_bits_storage_layout((unsigned char*)&s, sizeof(s));
```

编译执行后的输出结果为： `1000 1110 0000 0000 1010 0111 1100 0000`。可以看出Gcc采用了第二种方式，即在为a分配内存后，发现该存储单元剩余的空间(9 bits)已经无法存储下字段b了，于是乎Gcc又分配了一个存储单元(2个字节)用来为b分配空间，而a与b之间也因此存在了空隙。

我们还可以通过**匿名0长度位域字段**的语法强制位域在下一个存储单元开始分配，例如：

```
struct short_flag_t {
    unsigned short a : 2,
                   b : 3;
};
```

这个结构体本来是完全可以在一个存储单元(2字节)内为a、b两个位域分配空间的。如果我们非要让b放在与a不同的存储单元中，我们可以通过加入 匿名0长度位域的方法来实现：

```
struct short_flag_t {
    unsigned short a : 2;
    unsigned short   : 0;
    unsigned short b : 3;
};
```

这样声明后，sizeof(struct short_flag_t)变成了4。

```
struct short_flag_t s;
memset(&s, 0, sizeof(s));
s.a = 2; /* 10 */
s.b = 4; /* 100 */

dump_native_bits_storage_layout((unsigned char*)&s, sizeof(s));
```

执行后，输出的结果为：

```
0100 0000 0000 0000 0010 0000 0000 0000
```

可以看到位域b被强制放到了第二个存储单元中。如果没有那个匿名0长度的位域，那结果应该是这样的：

```
0100 1000 0000 0000
```

最后位域的长度是不允许超出其类型的最大长度的，比如：

```
struct short_flag_t {
    short a : 17;
};

error: width of ‘a’ exceeds its type
```

二、位域的位序

再回顾一下上一节的最后那个例子（不使用匿名0长度位域时）：

```
struct short_flag_t s;
memset(&s, 0, sizeof(s));
s.a = 2; /* 10 */
s.b = 4; /* 100 */
```

dump bits的结果为`0100 1000 0000 0000`。

怎么感觉输出的结果与s.a和s.b的值对不上啊！根据a和b的值，dump bits的输出似乎应该为1010 0000 0000 0000。对比这两个dump结果不同的部分：1010 0000 vs. 0100 1000，a和b的bit顺序恰好相反。之前一直与字节序做斗争，难不成bit也有序之分？事实就是这样的。bit也有order的概念，称为位序。位域字 段的内存位排序就称为该位域的位序。

我们来回顾一下字节序的概念，字节序分大端(big-endian，典型体系Sun Sparc)和小端(little-endian，典型体系Intel x86)：
大端指的是数值（比如0×12345678）的逻辑最高位(0×12)放在起始地址（低地址）上，简称高位低址，就是**高位放在起始地址。**
小端指的是数值（比如0×12345678）的逻辑最低位(0×78)放在起始地址（低地址）上，简称低位低址，就是**低位放在起始地址。**

看下面例子：

```
int
main()
{
    char c[4];
    unsigned int i = 0×12345678;
    memcpy(c, &i, sizeof(i));

    printf("%p - 0x%x\n", &c[0], c[0]);
    printf("%p - 0x%x\n", &c[1], c[1]);
    printf("%p - 0x%x\n", &c[2], c[2]);
    printf("%p - 0x%x\n", &c[3], c[3]);
}
```

在x86 (小端机器)上输出结果如下：

```
0x7fff1a6747c0 - 0×78
0x7fff1a6747c1 - 0×56
0x7fff1a6747c2 - 0×34
0x7fff1a6747c3 - 0×12
```

在sparc(大端机器)上输出结果如下：

```
ffbffbd0 - 0×12
ffbffbd1 - 0×34
ffbffbd2 - 0×56
ffbffbd3 - 0×78
```

通过以上输出结果可以看出，小端机器的数值低位0×78放在了低地址0x7fff1a6747c0上；而大端机器则是将数值高位0×12放在了低 地址0xffbffbd0上。

机器的最小寻址单位是字节，bit无法寻址，也就没有高低地址和起始地址的概念，我们需要定义一下bit的“地址”。以一个字节为例，我们把从左到右的8个bit的位置(position)命名按顺序命名如下：

```
p7 p6 p5 p4 p3 p2 p1 p0
```

其中最左端的p7为起始地址。这样以**一字节大小**的数值10110101(b)为例，其在不同平台下的内存位序如下：

大端的含义是数值的最高位1（最左边的1）放在了起始位置p7上，即数值10110101的大端内存布局为10110101。
小端的含义是数值的最低位1(最右边的1)放在了起始位置p7上，即数值10110101的小端内存布局为10101101。

前面的函数dump_native_bits_storage_layout也是符合这一定义的，即最左为起始位置。

同理，对于一个bit个数为3且存储的数值为110(b)的位域而言，将其3个bit的位置按顺序命名如下：

```
p2 p1 p0
```

其在大端机器上的bit内存布局，即位域位序为： 110;
其在小端机器上的bit内存布局，即位域位序为： 011。

在此基础上，理解上面例子中的疑惑就很简单了。

```
 s.a = 2; /* 10(b) ，大端机器上位域位序为 10，小端为01 */
 s.b = 4; /* 100(b)，大端机器上位域位序为100，小端为001 */
```

于是在x86（小端）上的dump bits结果为：0100 1000 0000 0000
而在sparc(大端）上的dump bits结果为：1010 0000 0000 0000

同时我们可以看出这里是根据位域进行单独赋值的，这样**位域的位序是也是以位域为单位排列的，即每个位域内部独立排序**， 而不是按照存储单元（这里的存储单元是16bit）或按字节内bit序排列的。

三、tcphdr定义分析

前面提到过在linux-kernel.cn mail list中的那个问题大致如下：

tcphdr定义中的大端代码：

```
__u16   doff:4,
        res1:4,
        cwr:1,
        ece:1,
        urg:1,
        ack:1,
        psh:1,
        rst:1,
        syn:1,
        fin:1;
```

问题是其对应的小端代码该如何做字段排序？似乎有两种方案摆在面前：

方案1:

```
__u16    res1:4,
         doff:4,
         fin:1,
         syn:1,
         rst:1,
         psh:1,
         ack:1,
         urg:1,
         ece:1,
         cwr:1;

 or
```

方案2：

```
__u16   cwr:1,
        ece:1,
        urg:1,
        ack:1,
        psh:1,
        rst:1,
        syn:1,
        fin:1,
        res1:4
        doff:4;
```

个人觉得这两种方案从理论上都是没错的，关键还是看tcphdr是如何进行pack的，是按__u16整体打包，还是按byte打包。原代码中使用的是方 案1，推测出tcphdr采用的是按byte打包的方式，这样我们只需调换byte内的bit顺序即可。res1和doff是一个字节内的两个位域，如果 按自己打包，他们两个的顺序对调即可在不同端的平台上得到相同的结果。用下面实例解释一下：

假设在大端系统上，doff和res1的值如下：

```
doff res1
1100 1010 大端
```

在大端系统上pack后，转化为网络序：

```
doff res1
1100 1010 网络序
```

小端系统接收后，转化为本地序：

```
0101 0011
```

很显然，我们应该按如下方法对应：

```
res1 doff
0101 0011
```

也就相当于将doff和res1的顺序对调，这样在小端上依旧可以得到相同的值。

Related posts:

1. GLIBC strlen源代码分析
2. 偿还N年前的一笔技术债
3. Go defer的C实现
4. 走马观花ANSI C标准-类型表示
5. Go与C语言的互操作