计算机中的内存对齐以及内存的堆栈分配原理

转载

空中海 于 2016-04-18 08:48:42 发布 1916 收藏 3

在大多数低层程序设计中，由于内存分配与内存对齐问题所带来的bug所占比重非常大。本文对内存分配中的分配空间类型、作用、方法、适用范围、优缺点以及内存对齐问题中的对齐原因、对齐规则等进行了详细的说明，并结合大量c语言代码进行阐述与分析。

[兹锴诓]

内存分配；堆栈原理；内存对齐；

1 引言

操作系统的内存分配问题与内存对齐问题对于低层程序设计来说是非常重要的，对内存分配的理解直接影响到代码质量、正确率、效率以及程序员对内存使用情况、溢出、泄露等的判断力。而内存对齐是常常被忽略的问题，理解内存对齐原理及方法则有助于帮助程序员判断访问非法内存。

2 程序的内存分配问题

一、一般C/C++程序占用的内存主要分为5种

1、栈区（stack）：类似于堆栈，由程序自动创建、自动释放。函数参数、局部变量以及返回点等信息都存于其中。

2、堆区（heap）： 使用自由，不需预先确定大小。多数情况下需要由程序员手动申请、释放。如不释放，程序结束后由操作系统垃圾回收机制收回。

3、全局区/静态区（static）：全局变量和静态变量的存储是区域。程序结束后由系统释放。

4、文字常量区：常量字符串就是放在这里的。 程序结束后由系统释放。

5、程序代码区：既可执行代码。

例：

```
#include
int quanju;
void fun(int f_jubu);
int main(void)
{
    int m_jubu;
    static int m_jingtai;
    char *m_zifum,*m_zifuc = "hello";
    void (*pfun)(int);
    pfun=&fun;
    m_zifum = (char *)malloc(sizeof(char)*10);
    pfun(1);
    printf("&quanju  : %x/n",&quanju);
    printf("&m_jubu  : %x/n",&m_jubu);
    printf("&m_jingtai: %x/n",&m_jingtai);
    printf("m_zifuc  : %x/n",m_zifuc);
    printf("&m_zifuc  : %x/n",&m_zifuc);
    printf("m_zifum  : %x/n",m_zifum);
    printf("&m_zifum  : %x/n",&m_zifum);
    printf("pfun     : %x/n",pfun);
    printf("&pfun    : %x/n",&pfun);
    getch();
    return 0;
```

```
}
void fun(int f_jubu)
{
    static int f_jingtai;
    printf("&f_jingtai: %x/n",&f_jingtai);
    printf("&f_jubu   : %x/n",&f_jubu);
}
```

输出结果：

```
&f_jingtai: 404020
&f_jubu   : 22ff40
&quanju   : 404070
&m_jubu   : 22ff74
&m_jingtai: 404010
m_zifuc   : 403000
&m_zifuc  : 22ff6c
m_zifum   : 3d24e0
&m_zifum  : 22ff70
pfun      : 4013af
&pfun     : 22ff68
```

分析：

堆区:

    m_zifum   : 3d24e0

代码区:

    pfun   : 4013af

局区/静态区（static）:

    m_zifuc   : 403000
    &m_jingtai: 404010
    &f_jingtai: 404020
    &quanju   : 404070

栈区:

    &f_jubu   : 22ff40 fun函数栈区
    &pfun   : 22ff68 主函数栈区
    &m_zifuc : 22ff6c
    &m_zifum : 22ff70
    &m_jubu   : 22ff74

二、堆和栈

    1申请方式

stack:

    由系统自动分配。 例如，声明在函数中一个局部变量 int b; 系统自动在栈中为b开辟空间

heap:

    需要程序员手动申请，并指明大小，在c中，有malloc函数完成

如p1 = (char *)malloc(10);

在C++中用new运算符

如p2 = (char *)malloc(10);

但是注意p1、p2本身是在栈中的。

    2 申请后系统的响应

栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

堆：大多数操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的free函数才能正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。

3申请大小的限制

栈：在Windows下,栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在WINDOWS下，栈的大小是2M（也有的说是1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示overflow。因此，能从栈获得的空间较小。

堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

4申请效率的比较：

栈由系统自动分配，速度较快。但程序员是无法控制的。

堆是由程序员手动分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便.

另外，在WINDOWS下，最好的方式是用VirtualAlloc分配内存，他不是在堆，也不是在栈是直接在进程的地址空间中保留一快内存，虽然用起来最不方便。但是速度快，也最灵活。

5堆和栈中的存储内容

栈： 在函数调用时，第一个进栈的是函数调用语句的下一条可执行语句的地址，然后是函数的各个参数，在大多数的C编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意静态变量是不入栈的。 当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是函数中的下一条指令，程序由该点继续运行。

堆：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容由程序员安排。

6存取效率的比较

```
char s1[] = "aaaaaaaaaaaaaa";
char *s2 = "bbbbbbbbbbbbbbbb";
aaaaaaaaaa是在运行时刻赋值的；
而bbbbbbbbbb是在编译时就确定的；
但是，在以后的存取中，在栈上的数组比指针所指向的字符串(例如堆)快。
比如：
#include
void main()
{
char a = 1;
char c[] = "1234567890";
char *p ="1234567890";
a = c[1];
a = p[1];
return;
}
对应的汇编代码
: a = c[1];
00401067 8A 4D F1 mov cl,byte ptr [ebp-0Fh]
0040106A 88 4D FC mov byte ptr [ebp-4],cl
: a = p[1];
0040106D 8B 55 EC mov edx,dword ptr [ebp-14h]
00401070 8A 42 01 mov al,byte ptr [edx+1]
00401073 88 45 FC mov byte ptr [ebp-4],al
```

第一种在读取时直接就把字符串中的元素读到寄存器cl中，而第二种则要先把指针值读到edx中，在根据edx读取字符，显然慢了一些。

2 内存对齐问题

一、内存对齐的原因

大部分的参考资料都是如是说的：

1、平台原因(移植原因)：不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。

2、性能原因：数据结构(尤其是栈)应该尽可能地在自然边界上对齐。原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；而对齐的内存访问仅需要一次访问。

二、对齐规则

每个特定平台上的编译器都有自己的默认“对齐系数”(也叫对齐模数)。程序员可以通过预编译命令#pragma pack(n)，n=1,2,4,8,16来改变这一系数，其中的n就是你要指定的“对齐系数”。

规则：

1、数据成员对齐规则：结构(struct)(或联合(union))的数据成员，第一个数据成员放在offset为0的地方，以后每个数据成员的对齐按照#pragma pack指定的数值和这个数据成员

自身长度中，比较小的那个进行。

2、结构(或联合)的整体对齐规则：在数据成员完成各自对齐之后，结构(或联合)本身也要进行对齐，对齐将按照#pragma pack指定的数值和结构(或联合)最大数据成员长度中，比较小的那个进行。

3、结合1、2可推断：当#pragma pack的n值等于或超过所有数据成员长度的时候，这个n值的大小将不产生任何效果。

三、试验

下面我们通过一系列例子的详细说明来证明这个规则

编译器：GCC 3.4.2、VC6.0

平台：Windows XP

典型的struct对齐

struct定义：

```
#pragma pack(n)
struct test_t {
 int a;
 char b;
 short c;
 char d;
};
#pragma pack(n)
```

首先确认在试验平台上的各个类型的size，经验证两个编译器的输出均为：

sizeof(char) = 1

sizeof(short) = 2

sizeof(int) = 4

试验过程如下：通过#pragma pack(n)改变“对齐系数”，然后察看sizeof(struct test_t)的值。

1、1字节对齐(#pragma pack(1))

输出结果：sizeof(struct test_t) = 8 [两个编译器输出一致]

分析过程：

1) 成员数据对齐

```
#pragma pack(1)
struct test_t {
```

```
 int a;
 char b;
 short c;
 char d;
};
#pragma pack()
```

成员总大小=8

2) 整体对齐

整体对齐系数 = min((max(int,short,char), 1) = 1

整体大小(size)=$(成员总大小) 按 $(整体对齐系数) 圆整 = 8 [注1]

2、2字节对齐(#pragma pack(2))

输出结果：sizeof(struct test_t) = 10 [两个编译器输出一致]

分析过程：

1) 成员数据对齐

```
#pragma pack(2)
struct test_t {
 int a;
 char b;
 short c;
 char d;
};
#pragma pack()
```

成员总大小=9

2) 整体对齐

整体对齐系数 = min((max(int,short,char), 2) = 2

整体大小(size)=$(成员总大小) 按 $(整体对齐系数) 圆整 = 10

3、4字节对齐(#pragma pack(4))

输出结果：sizeof(struct test_t) = 12 [两个编译器输出一致]

分析过程：

1) 成员数据对齐

```
#pragma pack(4)
struct test_t {
 int a;
 char b;
 short c;
 char d;
};
#pragma pack()
```

成员总大小=9

2) 整体对齐

整体对齐系数 = min((max(int,short,char), 4) = 4

整体大小(size)=$(成员总大小) 按 $(整体对齐系数) 圆整 = 12

4、8字节对齐(#pragma pack(8))

输出结果：sizeof(struct test_t) = 12 [两个编译器输出一致]

分析过程：

1) 成员数据对齐

```
#pragma pack(8)
struct test_t {
 int a;
 char b;
 short c;
 char d;
};
#pragma pack()
```

成员总大小=9

2) 整体对齐

整体对齐系数 = min((max(int,short,char), 8) = 4

整体大小(size)=$(成员总大小) 按 $(整体对齐系数) 圆整 = 12

5、16字节对齐(#pragma pack(16))

输出结果：sizeof(struct test_t) = 12 [两个编译器输出一致]

分析过程：

1) 成员数据对齐

```
#pragma pack(16)
struct test_t {
 int a;
 char b;
 short c;
 char d;
};
#pragma pack()
```

成员总大小=9

2) 整体对齐

整体对齐系数 = min((max(int,short,char), 16) = 4

整体大小(size)=$(成员总大小) 按 $(整体对齐系数) 圆整 = 12

8字节和16字节对齐试验证明了“规则”的第3点：“当#pragma pack的n值等于或超过所有数据成员长度的时候，这个n值的大小将不产生任何效果”。

4结束语

内存分配与内存对齐是个很复杂的东西，不但与具体实现密切相关，而且在不同的操作系统，编译器或硬件平台上规则也不尽相同，虽然目前大多数系统/语言都具有自动管理、分配并隐藏低层操作的功能，使得应用程序编写大为简单，程序员不在需要考虑详细的内存分配问题。但是，在系统或驱动级以至于高实时，高保密性的程序开发过程中，程序内存分配问题仍旧是保证整个程序稳定，安全，高效的基础。

一、什么是字节对齐,为什么要对齐?

现代计算机中内存空间都是按照byte划分的，从理论上讲似乎对任何类型的变量的访问可以从任何地址开始，但实际情况是在访问特定类型变量的时候经常在特 定的内存地址访问，这就需要各种类型数据按照一定的规则在空间上排列，而不是顺序的一个接一个的排放，这就是对齐。

对齐的作用和原因：各个硬件平台对存储空间的处理上有很大的不同。一些平台对某些特定类型的数据只能从某些特定地址开始存取。比如有些架构的CPU在访问 一个没有进行对齐的变量的时候会发生错误,那么在这种架构下编程必须保证字节对齐.其他平台可能没有这种情况，但是最常见的是如果不按照适合其平台要求对 数据存放进行对齐，会在存取效率上带来损失。比如有些平台每次读都是从偶地址开始，如果一个int型（假设为32位系统）如果存放在偶地址开始的地方，那 么一个读周期就可以读出这32bit，而如果存放在奇地址开始的地方，就需要2个读周期，并对两次读出的结果的高低字节进行拼凑才能得到该32bit数 据。显然在读取效率上下降很多。

二、请看下面的结构：

```
struct MyStruct  {

        double dda1;

        char dda;

        int type  };
```

对结构MyStruct采用sizeof会出现什么结果呢？sizeof(MyStruct)为多少呢？

也许你会这样求：

sizeof(MyStruct)=sizeof(double)+sizeof(char)+sizeof(int)=13

但是当在VC中测试上面结构的大小时，你会发现sizeof(MyStruct)为16。你知道为什么在VC中会得出这样一个结果吗？其实，这是VC对变量存储的一个特殊处理。为了提高CPU的存储速度，VC对一些变量的起始地址做了“对齐”处理。在默认情况下，VC规定各成员变量存放的起始地址相对于结构的起始地址的偏移量必须为该变量的类型所占用的字节数的倍数。

下面列出常用类型的对齐方式(vc6.0,32位系统)。

类型　对齐方式（变量存放的起始地址相对于结构的起始地址的偏移量）

Char　偏移量必须为sizeof(char)即1的倍数

int　偏移量必须为sizeof(int)即4的倍数

float　偏移量必须为sizeof(float)即4的倍数

double　偏移量必须为sizeof(double)即8的倍数

Short　偏移量必须为sizeof(short)即2的倍数

各成员变量在存放的时候根据在结构中出现的顺序依次申请空间，同时按照上面的对齐方式调整位置，空缺的字节VC会自动填充。同时VC为了确保结构的大小为结构的字节边界数（即该结构中占用最大空间的类型所占用的字节数）的倍数，所以在为最后一个成员变量申请空间后，还会根据需要自动填充空缺的字节。(填充的时候根据结构体中的最大的比如(double型是最大的)的整数倍)