

1 保护模式

现在操作系统大多是运行在保护模式下的，如果我们要学好操作系统，保护模式是一定要了解的；本章节主要讲解x86的保护模式，由于x64是基于x86拓展的指令集，所以当你具备x86的基础之后再去学习x64也就很简单了。

保护模式的特点就是段、页的机制，这也是保护模式所需要学习的内容。学习保护模式才能真正的理解内核是如何运作的。

2 段

2.1 段寄存器

如果你学习过初级篇的课程，实际你就已经使用过段寄存器了，如下汇编代码：

```
1  mov dword ptr ds:[0x123456], eax
```

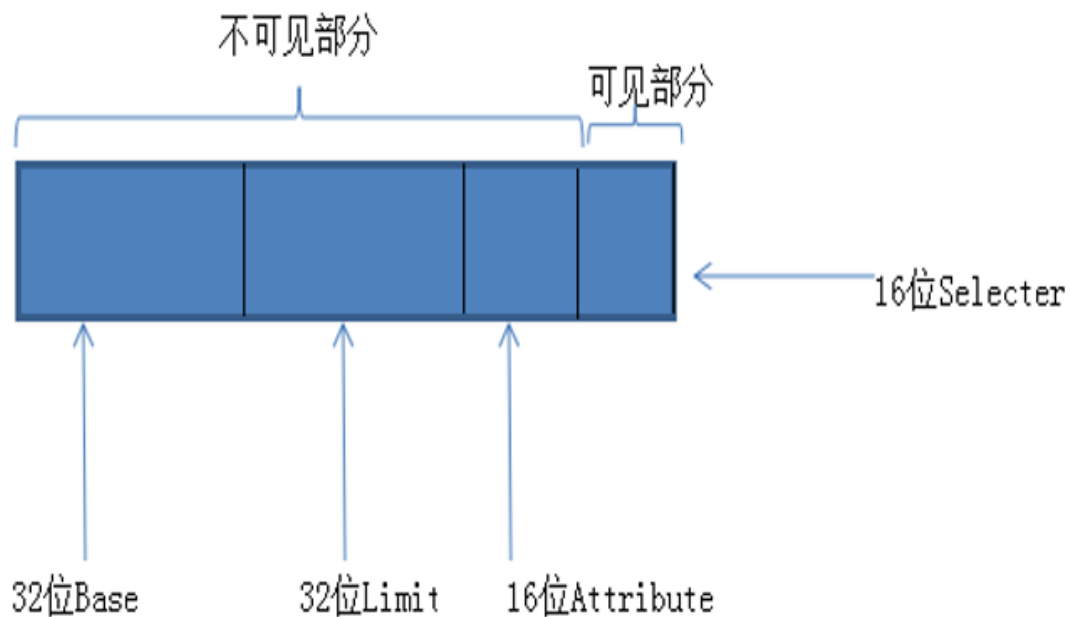
这是向某一地址写入内容，汇编代码中的DS实际上就是段寄存器，真正写入的地址实际上是 **DS.base+0x123456**，base就是DS段寄存器的一个属性。（其他段寄存器的形式也是如此。）

段寄存器一共有8个：ES、CS、SS、DS、FS、GS、LDTR、TR。

2.1.1 结构

段寄存器的结构与我们之前所了解的通用寄存器结构是不一样的，通用寄存器有32位（4字节）宽度，但是段寄存器却有96位（12字节）宽度。

段寄存器的结构如下图所示，虽然它有96位宽度，但我们可见的部分只有16位：



用结构体来表示就是如下图所示，有16位可见部分（Selector），16位属性，32位Base和32位Limit：

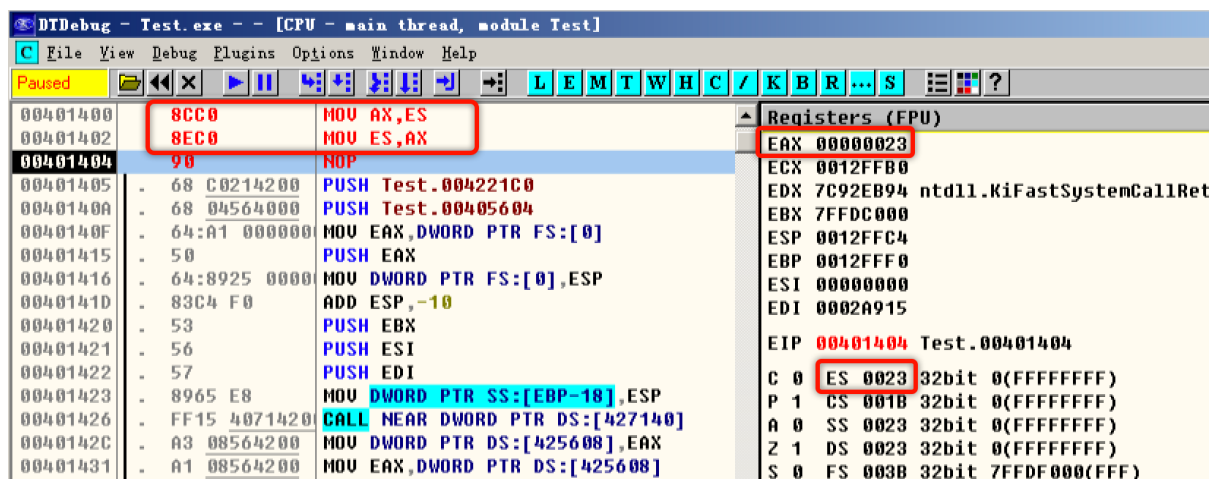
```
1  struct SegMent {
2      WORD Selector;
3      WORD Attributes;
4      DWORD Base;
5      DWORD Limit;
```

```
6    };
```

16位属性指的是当前段寄存器的可读、可写的属性；32位Base表示当前段是从哪里开始的；32位Limit表示当前段的整个长度。

2.1.2 读写

由于我们只能看见段寄存器的16位，所以我们也只能对段寄存器的这可见部分进行读取，但写入的话是按照96位去写入的。



如上图所示我们可以使用MOV指令对段寄存器进行读写，但是这2个段寄存器：LDTR、TR，它们是不可以使用MOV指令进行读写的。

2.1.3 属性探测

上文中我们知道段寄存器的结构中是有几个成员（属性）的，我们可见的部分是16位的Selector，其余的都无法看见，那么这些属性都是真实存在的还是一个虚无缥缈的概念呢，我们可以使用代码来证实这些属性的存在。

如下表所示，是我当前Windows XP虚拟机环境中的段寄存器相关的属性，标红部分即表示这些属性对应的值是不固定的：

| 段寄存器 | Selector | Attribute | Base | Limit |
|------|----------|-----------|-----------|------------|
| ES | 0023 | 可读、可写 | 0 | 0xFFFFFFFF |
| CS | 001B | 可读、可执行 | 0 | 0xFFFFFFFF |
| SS | 0023 | 可读、可写 | 0 | 0xFFFFFFFF |
| DS | 0023 | 可读、可写 | 0 | 0xFFFFFFFF |
| FS | 003B | 可读、可写 | 0x7FDF000 | 0xFFF |
| GS | - | - | - | - |

如上表格这些属性值都可以在调试器的寄存器窗口中找到：

```

Registers (FPU)
EAX 00000023
ECX 0012FFB0
EDX 7C92EB94 ntdll.KiFastSystemCallRet
EBX 7FFDC000
ESP 0012FFC4
EBP 0012FFF0
ESI 00000000
EDI 0002A915
EIP 00401404 Test.00401404

C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FDF000(FFF)
T 0 GS 0000 NULL
D 0
0 0 LastErr ERROR_SUCCESS (00000000)

```

注：GS段寄存器没有属性值是因为Windows XP中并没有使用到它。

探测Attribute

首先我们来探测Attribute属性，如上表中，段寄存器CS的Attribute是可读、可执行，也就表示我们没法对该段寄存器进行写入，所以我们可以通过如下代码去探测它，该代码的意义就是将段寄存器CS的值读到AX寄存器中，再将AX寄存器的值读入到DS段寄存器中，最后通过DS段寄存器去写入内容：

```

1  void main()
2  {
3      int var = 0;
4      __asm {
5          MOV AX,CS
6          MOV DS,AX
7          MOV DWORD PTR DS:[var], eax
8      }
9  }

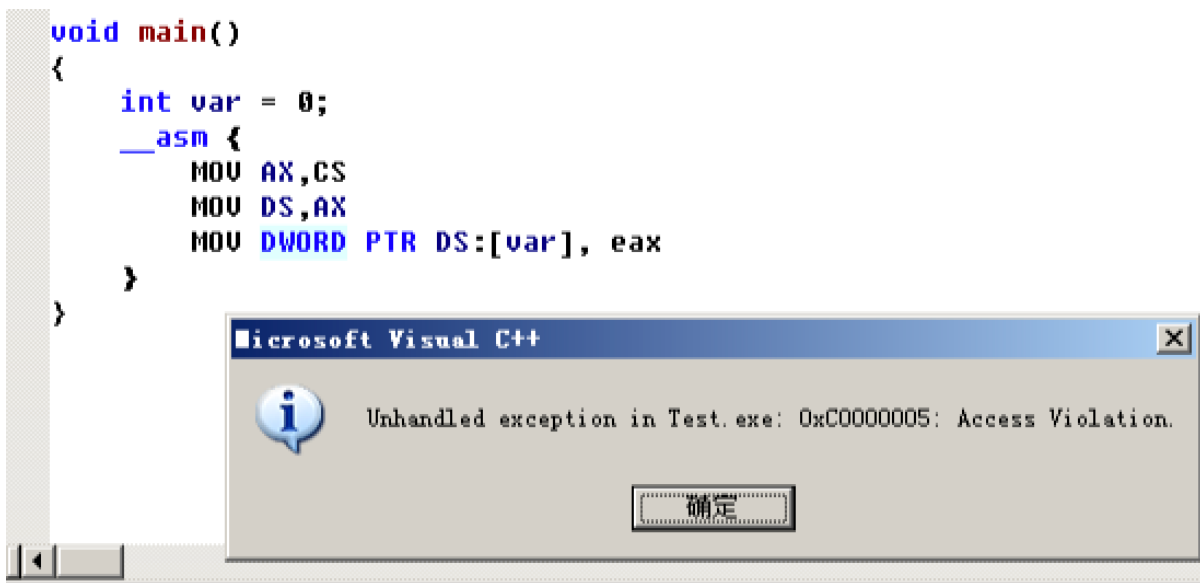
```

```

7:      int var = 0;
00401028    mov     dword ptr [ebp-4],0
8:      __asm {
9:          MOV AX,CS
0040102F    mov     ax,cs
10:         MOV DS,AX
00401032    mov     ds,ax
11:         MOV DWORD PTR DS:[var], eax
00401035    mov     dword ptr [ebp-4],eax
12:         }
13:     }

```

如果这段代码可以执行则表示我们可以对段寄存器CS进行写入内容，反之则证明其Attribute属性的真实性。编译代码并运行可执行文件，最终提示我们出了问题，由此我们便探测出了段寄存器CS的Attribute属性：



探测Base

我们之前在了解段寄存器的时候提到，当你以如下这种汇编指令去向地址写值时，实际上写入的地址是段寄存器的Base属性加上0x123456：

```
1  mov dword ptr ds:[0x123456], eax
```

因此，我们想要去探测段寄存器的Base属性，可以使用向0x0地址写值，因为我们都知道正常情况下0x0地址是不可读不可写的，但是FS段寄存器的Base属性值为0x7FFDF000，所以我们可以使用如下代码去探测：

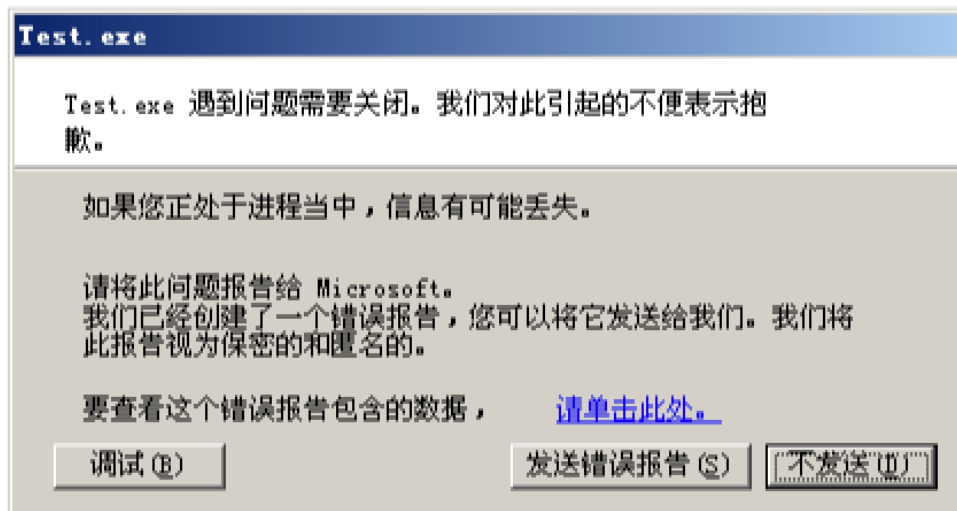
```
1  void main()
2  {
3      int var = 0;
4      __asm {
5          MOV AX,FS
6          MOV GS,AX
7          // MOV DWORD PTR DS:[0], EAX
8          MOV DWORD PTR GS:[0], EAX
9      }
10 }
```

如上代码编译并运行不会有任何报错，因为我们将EAX寄存器值写入到的实际地址是0x7FFDF000+0x0，如若你将MOV指令后的GS换成DS则编译运行就会报错：

```

void main()
{
    int var = 0;
    __asm {
        MOV AX,FS
        MOV GS,AX
        MOV DWORD PTR DS:[0], EAX
        // MOV DWORD PTR GS:[0], EAX
    }
}

```



探测Limit

我们知道Limit表示当前段的整个长度，FS段寄存器的Limit属性值为0xFFFF，因此我们可以设定一个超出长度的值去写入来探测Limit属性：

```

1 void main()
2 {
3     int var = 0;
4     __asm {
5         MOV AX,FS
6         MOV GS,AX
7         MOV DWORD PTR GS:[0x1000], EAX
8     }
9 }

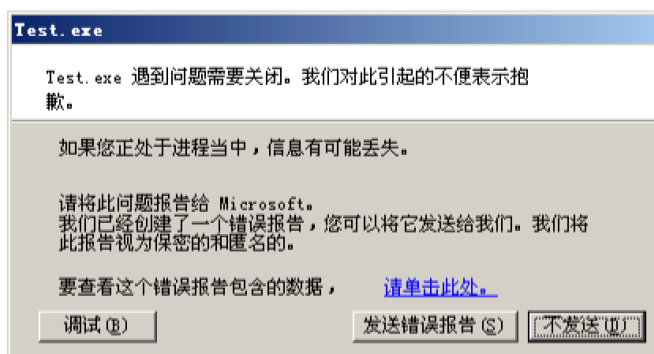
```

如上代码中，我们向0x7FFDF000+0x1000写入内容，由于FS段的长度只有0xFFFF，0x1000很明显大于0xFFFF，所以这段代码编译运行就会报错：

```

void main()
{
    int var = 0;
    __asm {
        MOV AX,FS
        MOV GS,AX
        MOV DWORD PTR GS:[0x1000], EAX
    }
}

```



2.2 描述符与选择子

在之前的章节中我们使用了如下这种指令向段寄存器中写入值，并且在最初介绍段寄存器的时候就提到，虽然这里我们的是将16位寄存器的值写入进去，但实际上写入的仍然是96位：

| | |
|---|-----------|
| 1 | MOV GS,AX |
|---|-----------|

那么另外80位的值是从何而来的呢？我们就需要了解这两个概念：GDT（Global Descriptor Table，全局描述符表）、LDT（Local Descriptor Table，局部描述符表）。

当我们执行如上类似指令时，CPU会根据AX的值选择GDT、LDT这两张表中的一张表进行查询，在表中查询出对应的信息给到段寄存器。

2.2.1 双机调试环境

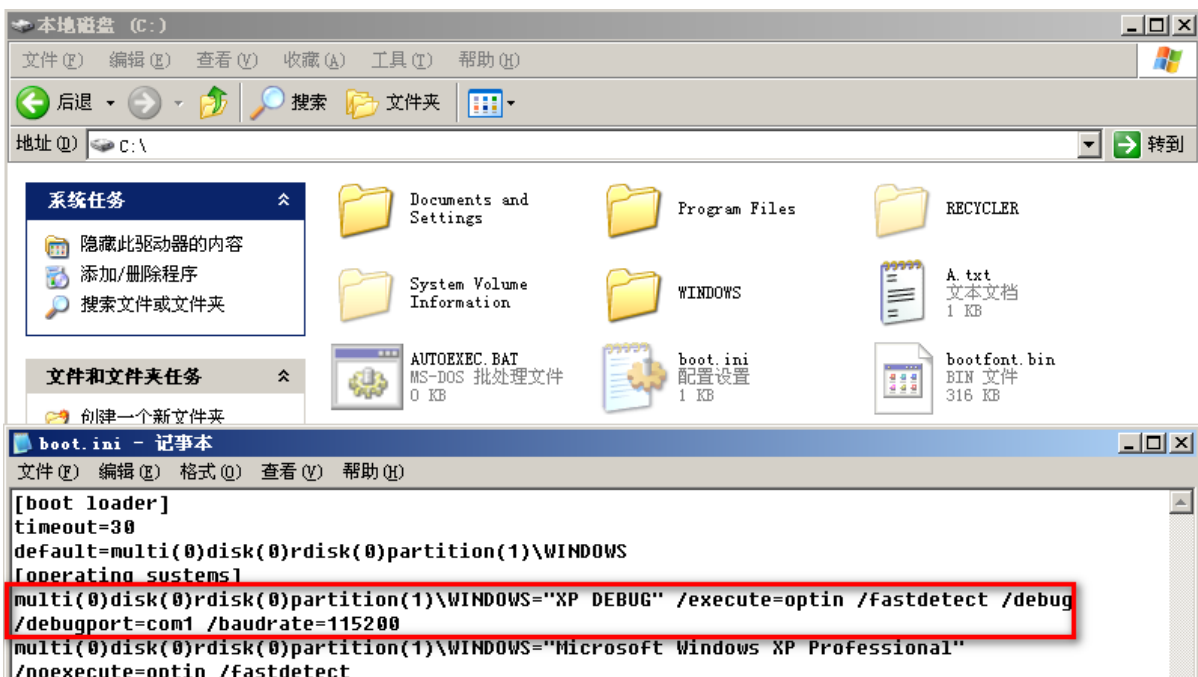
这里我们想要去查看GDT表就需要使用Windbg调试内核，为了便于未来的学习调试，我们需要来配置一下调试环境。调试模式为双机调试（因为你调试的是内核，本机调试的话，你在内核断点实际上整个系统也就停止了，你也就无法继续调试，所以我们需要使用双机调试模式进行调试。），你需要在你的物理机器上安装VMware虚拟机，接着在VMware中安装Windows XP系统的虚拟机；接着，在物理机器上安装Windbg，你可以通过微软官方地址进行下载安装：<https://docs.microsoft.com/zh-cn/windows-hardware/drivers/debugger/debugger-download-tools>。环境安装流程大致就是如此，这里不再进行图文赘述。

当你准备好环境之后，设置Window XP虚拟机，添加串行端口并按如下图进行配置：



然后在Windows XP虚拟机中修改C盘下的boot.ini文件（取消文件隐藏即可看见），添加如下内容即可：

| | |
|---|---|
| 1 | <code>multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="XP DEBUG" /execute=optin /fastdetect /debug / debugport=com1 /baudrate=115200</code> |
|---|---|



接着找到你物理机上的Windbg (x86) 文件，并发送快捷方式到桌面，修改快捷方式的目标值为：

```
1 "你的目录\windbg.exe" -b -k com:pipe,port=\\.\pipe\com_1,baud=115200,pipe
```



这时候我们的环境准备工作已经完成了，重启虚拟机至如下界面，选择**XP DEBUG**后用**调试程序**进入系统：

请选择要启动的操作系统：

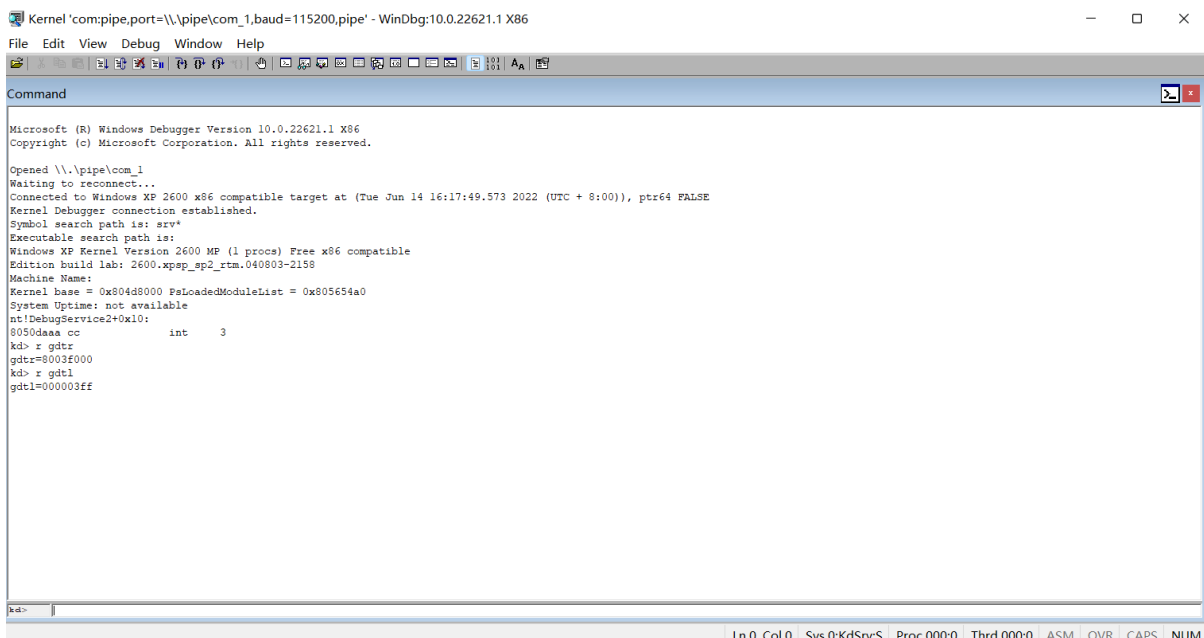
XP DEBUG [启用调试程序]

Microsoft Windows XP Professional

使用 ↑ 键和 ↓ 键来移动高亮显示条到所要的操作系统，
按 Enter 键做个选择。

要排解疑难以及了解 Windows 高级启动选项，请按 F8。

然后快速的双击你修改后的Windbg快捷方式，如下图所示我们就可以去观察Windows XP的内核了：



```

Kernel 'com:pipe,port=\\.\pipe\com_1,baud=115200,pipe' - WinDbg:10.0.22621.1 X86
File Edit View Debug Window Help
Microsoft (R) Windows Debugger Version 10.0.22621.1 X86
Copyright (c) Microsoft Corporation. All rights reserved.

Opened \\.\pipe\com_1
Waiting to reconnect...
Connected to Windows XP 2600 x86 compatible target at (Tue Jun 14 16:17:49.573 2022 (UTC + 8:00)), ptr64 FALSE
Kernel Debugger connection established.
Symbol search path is: srv*
Executable search path is:
Windows XP Kernel Version 2600 MP (1 procs) Free x86 compatible
Edition build lab: 2600.xpsp_sp2_rtm.040803-2158
Machine Name:
Kernel base = 0x804d8000 PsLoadedModuleList = 0x805654a0
System Uptime: not available
nt!DebugService2+0x10:
8050daaa cc          int     3
kd> r gdt
gdt=8003f000
kd> r gdt1
gdt1=0000003ff
kd>
Ln 0, Col 0 Sys 0:KdSrv:S Proc 000:0 Thrd 000:0 ASM OVR CAPS NUM

```

2.2.2 GDT表信息

我们可以使用Windbg去查看GDT表的一些信息，在Windbg中使用如下指令去查看GDT表相关的属性：

- 1 r gdtr - 查看GDT表的位置
- 2 r gdtl - 查看GDT表的大小
- 3 // gdtr、gdtl也都称之为寄存器

```
Opened \\.\pipe\com_1
Waiting to reconnect...
Connected to Windows XP 2600 x86 compatible target at (Tue Jun 14 16:17:49.573 2022 (UTC + 8:00)), ptr64 FALSE
Kernel Debugger connection established.
Symbol search path is: srv*
Executable search path is:
Windows XP Kernel Version 2600 MP (1 procs) Free x86 compatible
Edition build lab: 2600.xpsp_sp2_rtm.040803-2158
Machine Name:
Kernel base = 0x804d8000 PsLoadedModuleList = 0x805654a0
System Uptime: not available
nt!DebugService2+0x10:
8050daaa cc          int      3
kd> r gdtr
gdtr=8003f000
kd> r gdtl
gdtl=000003ff
```

我们知道GDT表的位置之后就可以使用如下指令去找到这张表：

- 1 dd gdtr寄存器的值

```
kd> dd 8003f000
ReadVirtual: 8003f000 not properly sign extended
8003f000  00000000 00000000 0000ffff 00cf9b00
8003f010  0000ffff 00cf9300 0000ffff 00cffa00
8003f020  0000ffff 00cff300 200020ab 80008b04
8003f030  f0000001 ffc093df 00000fff 0040f300
8003f040  0400ffff 0000f200 00000000 00000000
8003f050  97000068 80008955 97680068 80008955
8003f060  2f30ffff 00009302 80003fff 0000920b
8003f070  700003ff ff0092ff 0000ffff 80009a40
```

如上图，最左边的800开头的就是地址，右边的就是表中的数据。

这里我们使用dd指令去查看数据实际上不是很方便，因为它表示以4字节一组的方式展示数据，我们为了看起来更加方便我们可以使用dq指令去查看数据，它是以8字节一组的方式展示数据：

```
kd> dq 8003f000
8003f000  00000000`00000000 00cf9b00`0000ffff
8003f010  00cf9300`0000ffff 00cffa00`0000ffff
8003f020  00cff300`0000ffff 80008b04`200020ab
8003f030  ffc093df`f0000001 0040f300`00000fff
8003f040  0000f200`0400ffff 00000000`00000000
8003f050  80008955`97000068 80008955`97680068
8003f060  00009302`2f30ffff 0000920b`80003fff
8003f070  ff0092ff`700003ff 80009a40`0000ffff
```

我们通过gdtl寄存器知道当前GDT表的大小，但是使用dq/dd指令去查看表的时候很明显，它展示出来的数据并没有那么多，所以我们可以之后加上参数来自定义我们返回数据大小：

- | | |
|---|--|
| 1 | dq gdtl寄存器的值 L40 |
| 2 | // L不区分大小写，40在这里表示十六进制0x40，当前指令的意思是展示0x40组（dq->8字节）GDT表的数据 |

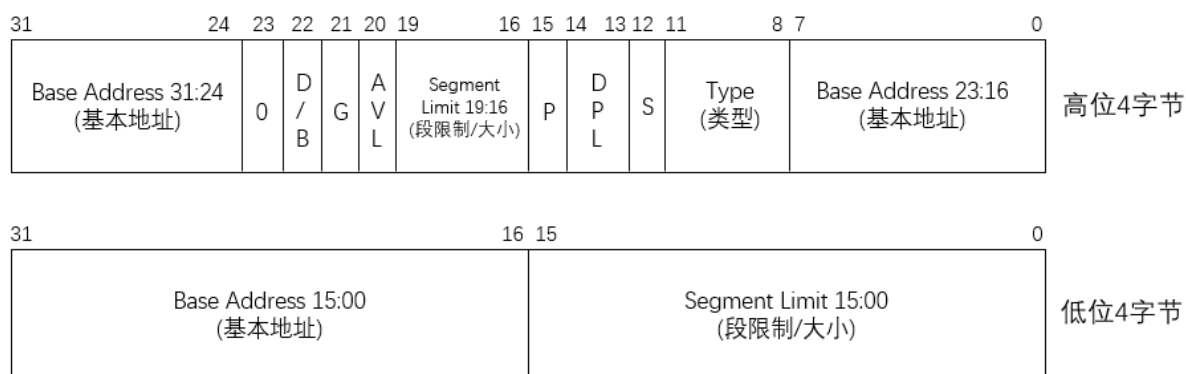
```

kd> dq 8003f000 L40
ReadVirtual: 8003f080 not properly sign extended
8003f000  00000000`00000000 00cf9b00`0000ffff
8003f010  00cf9300`0000ffff 00cffa00`0000ffff
8003f020  00cff300`0000ffff 80008b04`200020ab
8003f030  ffc093df`f0000001 0040f300`0000fff
8003f040  0000f200`0400ffff 00000000`00000000
8003f050  80008955`97000068 80008955`97680068
8003f060  00009302`2f30ffff 0000920b`80003fff
8003f070  ff0092ff`700003ff 80009a40`0000ffff
8003f080  80009240`0000ffff 00009200`00000000
8003f090  00000000`00000000 00000000`00000000
8003f0a0  00000000`00000000 00000000`00000000
8003f0b0  00000000`00000000 00000000`00000000
8003f0c0  00000000`00000000 00000000`00000000
8003f0d0  00000000`00000000 00000000`00000000
8003f0e0  00000000`8003f100 00009200`0000ffff
8003f0f0  8003984d`a798de0f 00009200`0000ffff
8003f100  00000000`8003f108 00000000`8003f110
8003f110  00000000`8003f118 00000000`8003f120
8003f120  00000000`8003f128 00000000`8003f130
8003f130  00000000`8003f138 00000000`8003f140
8003f140  00000000`8003f148 00000000`8003f150
8003f150  00000000`8003f158 00000000`8003f160
8003f160  00000000`8003f168 00000000`8003f170
8003f170  00000000`8003f178 00000000`8003f180
8003f180  00000000`8003f188 00000000`8003f190
8003f190  00000000`8003f198 00000000`8003f1a0
8003f1a0  00000000`8003f1a8 00000000`8003f1b0
8003f1b0  00000000`8003f1b8 00000000`8003f1c0
8003f1c0  00000000`8003f1c8 00000000`8003f1d0
8003f1d0  00000000`8003f1d8 00000000`8003f1e0
8003f1e0  00000000`8003f1e8 00000000`8003f1f0
8003f1f0  00000000`8003f1f8 00000000`8003f200

```

2.2.3 段描述符

在GDT表中存储的元素就是段描述符，每一个段描述符的宽度是8字节。段描述符的结构如下图所示：



这是一个从高到低的顺序，同时在Windbg中对应的也可以按图将数据平铺过来：

```
kd> dq 8003f000 L40
ReadVirtual: 8003f080 not properly sign extended
8003f000  00000000 00000000 00cf9b00`0000ffff
8003f010  00cf300`0000ffff 00cf9b00`0000ffff
8003f020  00cf300`0000ffff 80008b04`200020ab
8003f030  ffc093df`f0000001 0040f300`000000ff
8003f040  0000f200`0400ffff 00000000`00000000
8003f050  80008955`97000068 80008955`97680068
```

dq/dd指令会帮我们自动排好，便于对照着结构直接看，如果你想观察的仔细一些可以使用db指令：

```
kd> dd 8003f000 L4
8003f000  00000000 00000000 0000ffff 00cf9b00
kd> dq 8003f000 L2
8003f000  00000000`00000000 00cf9b00`0000ffff
kd> db 8003f008 L8
8003f008  ff ff 00 00 00 9b cf 00  ....
```

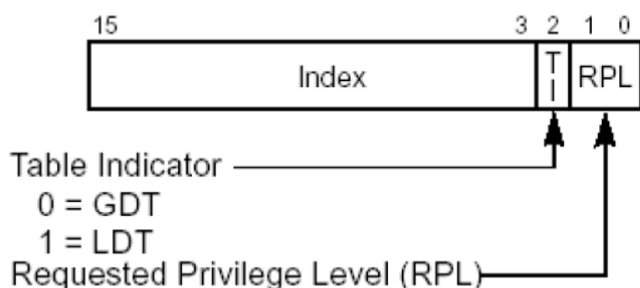
如下图所示这个对应关系就很容易理解了，其他的以此类推即可：

```
8003f008  ff ff 00 00 00 9b cf 00  ....
           Base Address Segment Limit
```

2.2.4 段选择子

段选择子是一个16位的段描述符，该描述符指向了定义该段的段描述符。之前我们所举例的AX寄存器就可以称之为段选择子。

如下图所示就是段选择子的结构及对应位的属性说明：



RPL: 请求特权级别

TI:
 TI=0 查GDT表
 TI=1 查LDT表

Index:
 处理器将索引值乘以8在加上GDT或者LDT的基地址，就是要加载的段描述符。

所以我们可以通过段选择子的TI位知道查询GDT还是LDT表（**注意，在Windows XP中没有使用LDT表，所以默认情况下TI位都是0。**），Index位可以知道具体要查的表的哪一个段描述符。

2.2.5 加载段描述符至段寄存器

之前我们所举例的MOV指令本质上就是加载段描述符至段寄存器，除了该指令外，我们还可以使用LES、LSS、LDS、LFS、LGS指令完成这一动作。

需要注意的是，CS段寄存器不能通过上述指令进行修改，这是因为CS为代码段，CS的改变会导致EIP的改变；如果你想要修改CS，就需要保证CS和EIP一起修改，这个在之后的章节中会学到。

如下代码所示，我们使用LES指令，它的意思是将buffer的高2个字节（段选择子）给到ES段寄存器，低4个字节给到ECX寄存器：

```

1 void main()
2 {
3     char buffer[6];
4     __asm {
5         LES ECX, FWORD PTR DS:[buffer]
6     }
7 }
```

需要注意的是段选择子的RPL数值要小于段描述符中的DPL。

2.2.6 段描述符的属性

当我们执行MOV类似指令时，CPU会根据段选择子的值选择GDT、LDT这两张表中的一张表进行查询，在表中查询出段描述符给到段寄存器，**但是我们知道段描述符只有64位，而除了段选择子本身的值外，段寄存器还需要80位，所以这里64位是如何变成80位的呢？**这就需要我们去了解段描述符的相关属性。

P位

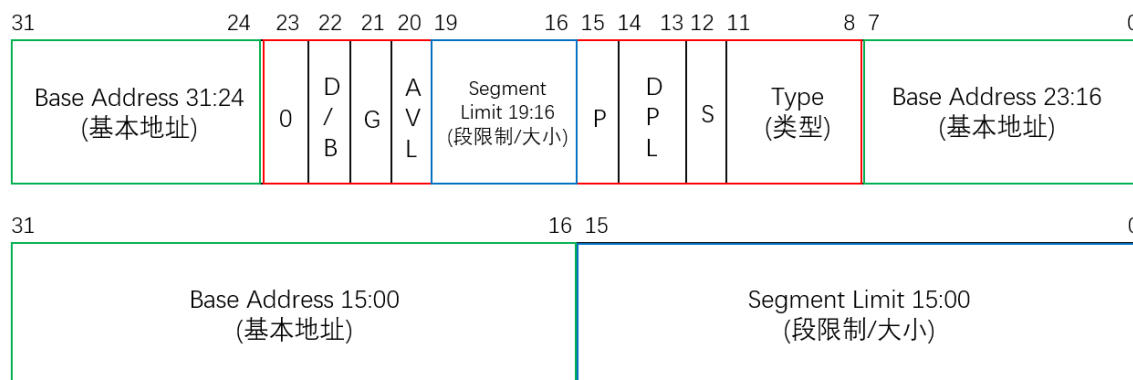
P位是段描述符高位4字节的第15位，它的值决定了段描述符是否是有效的。

```

1 P = 1 -> 段描述符有效
2 P = 0 -> 段描述符无效
```


G位

在了解G位属性之前，我们先来看一下段描述符与段寄存器结构的对应关系，如下图所示我使用颜色方框标记出了它们之间的对应关系（大家可以会觉得段描述符中的属性分布很零碎，这是因为段描述符也是一步一步发展的，Intel需要考虑老版本的系统就需要向下兼容，所以段描述符的结构就不能有变化，只能在以前的结构上进行拓展）：



```
struct Segment {
    WORD Selector; // 16位
    WORD Attributes; // 16位
    DWORD Base; // 32位
    DWORD Limit; // 32位
};
```

-> 高位4字节，第8位至第23位
 -> 三部分组成：低位4字节的第16位至31位、高位4字节的第0位至第7位、
 高位4字节的第24位至31位
 -> 两部分组成：低位4字节的第0位至第15位、高位4字节的第16位至第19位

仔细看这张图，你就会发现在段寄存器结构中的成员Limit对应到段描述符中只有20位，其余的12位没有了，也就表示这里的Limit最大值就是0xFFFFF，而另外12位取决于描述符的属性G位。

当G位的值为0时，Limit最大值就是0xFFFFF，按32位书写就是0x000FFFFF；当G位的值为1时，Limit的单位就是4KB，它的取值公式就是：（根据段描述符中拼接出来的Limit值（20位）+ 1）* 4KB - 1，根据取值公式此时Limit的最大值为0xFFFFFFFF。

至此，我们就知道了CPU是如何将段描述符的对应属性取出来放入段寄存器的结构中，也解释了64位变成80位的逻辑。

S位

S位是段描述符高位4字节的第12位，它的值表示当前描述符是属于什么段的：

- | | |
|---|-----------------------|
| 1 | S = 1 -> 表示代码段或数据段描述符 |
| 2 | S = 0 -> 表示系统段描述符 |

Type域

S位的值同样也决定了Type域的值，当S位的值为1时表示当前为代码或数据段描述符，那么Type域的值就是如下这张表格：

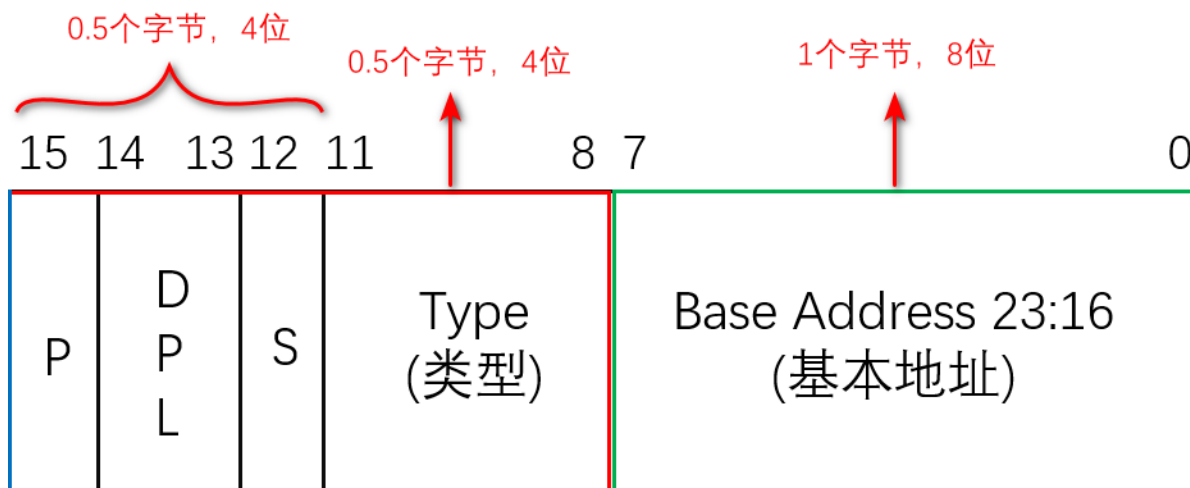
Table 3-1. Code- and Data-Segment Types

| Type Field | | | | | Descriptor Type | Description |
|------------|----|---------|--------|--------|-----------------|---|
| Decimal | 11 | 10 E | 9 W | 8 A | | |
| 0 | 0 | 0 | 0 | 0 | Data | Read-Only |
| 1 | 0 | 0 | 0 | 1 | Data | Read-Only, accessed |
| 2 | 0 | 0 | 1 | 0 | Data | Read/Write |
| 3 | 0 | 0 | 1 | 1 | Data | Read/Write, accessed |
| 4 | 0 | 1 | 0 | 0 | Data | Read-Only, expand-down |
| 5 | 0 | 1 | 0 | 1 | Data | Read-Only, expand-down, accessed |
| 6 | 0 | 1 | 1 | 0 | Data | Read/Write, expand-down |
| 7 | 0 | 1 | 1 | 1 | Data | Read/Write, expand-down, accessed |
| | | C | R | A | | |
| 8 | 1 | 0 | 0 | 0 | Code | Execute-Only |
| 9 | 1 | 0 | 0 | 1 | Code | Execute-Only, accessed |
| 10 | 1 | 0 | 1 | 0 | Code | Execute/Read |
| 11 | 1 | 0 | 1 | 1 | Code | Execute/Read, accessed |
| 12 | 1 | 1 | 0 | 0 | Code | Execute-Only, conforming |
| 13 | 1 | 1 | 0 | 1 | Code | Execute-Only, conforming, accessed |
| 14 | 1 | 1 | 1 | 0 | Code | Execute/Read-Only, conforming |
| 15 | 1 | 1 | 1 | 1 | Code | Execute/Read-Only, conforming, accessed |

如表格所示，Type域是段描述符高位4字节的第8位至第11位，第11位为0则是数据段，为1则是代码段。在详细了解这张表格之前，我们先来根据实际的GDT表来找一下数据段/代码段的描述符。

首先我们需要根据P位的值为1来确定描述符是有效的，接着DPL在Windows里只能出现两种情况：00/11（**后期章节中会讲解该属性**），然后S位必须是1才能是数据段/代码段的描述符，所以我们可以得出P、DPL、S拼接的值为：1001/1111，转为十六进制就是：0x9/0xF。

最后我们来看下Type域名的值，按表格所示，它的值也就是0x0 - 0xF，并且0x8是一个界限点，当值小于0x8则表示当前段是数据段，反之则是代码段。



综上所述，我们去看GDT表中的内容，如下图所示红色方框标记的就是代码段，绿色的数据段：

```

kd> dq 8003f000
8003f000  00000000`00000000 00cf9b00`0000ffff
8003f010  00cf9300`0000ffff 00cffa00`0000ffff
8003f020  00cff300`0000ffff 80008b04`200020ab
8003f030  ff093df`f0000001 0040f300`0000ffff
8003f040  0000f200`0400ffff 00000000`00000000
8003f050  80008955`97000068 80008955`97680068
8003f060  00009302`2f30ffff 0000920b`80003fff
8003f070  ff0092ff`700003ff 80009a40`0000ffff

```

我们回过头来再看Type域的表格，除了第11位外，第8、9、10位在代码和数据段中都有不同的意义。

在数据段时，**A**（访问位，第8位）表示该段是否被访问过，当处理器将该段描述符置入某个段寄存器时，其值就会被修改为1；**W**（可写位，第9位）表示当前数据段是否可写；**E**（拓展位，第10位），当该值为0时向上拓展，当该值为1时向下拓展。

在代码段时候，**A**（访问位，第8位）与数据段一样的意义；**R**（可读位，第9位）表示当前代码段是否可读；**C**（一致位，第10位），当该值为1时则表示一致代码段，当该值为0时表示非一致代码段。

Table 3-1. Code- and Data-Segment Types

| Type Field | | | | | Descriptor Type | Description |
|------------|----|----------|----------|----------|-----------------|---|
| Decimal | 11 | 10 E | 9 W | 8 A | | |
| 0 | 0 | 0 | 0 | 0 | Data | Read-Only |
| 1 | 0 | 0 | 0 | 1 | Data | Read-Only, accessed |
| 2 | 0 | 0 | 1 | 0 | Data | Read/Write |
| 3 | 0 | 0 | 1 | 1 | Data | Read/Write, accessed |
| 4 | 0 | 1 | 0 | 0 | Data | Read-Only, expand-down |
| 5 | 0 | 1 | 0 | 1 | Data | Read-Only, expand-down, accessed |
| 6 | 0 | 1 | 1 | 0 | Data | Read/Write, expand-down |
| 7 | 0 | 1 | 1 | 1 | Data | Read/Write, expand-down, accessed |
| | | C | R | A | | |
| 8 | 1 | 0 | 0 | 0 | Code | Execute-Only |
| 9 | 1 | 0 | 0 | 1 | Code | Execute-Only, accessed |
| 10 | 1 | 0 | 1 | 0 | Code | Execute/Read |
| 11 | 1 | 0 | 1 | 1 | Code | Execute/Read, accessed |
| 12 | 1 | 1 | 0 | 0 | Code | Execute-Only, conforming |
| 13 | 1 | 1 | 0 | 1 | Code | Execute-Only, conforming, accessed |
| 14 | 1 | 1 | 1 | 0 | Code | Execute/Read-Only, conforming |
| 15 | 1 | 1 | 1 | 1 | Code | Execute/Read-Only, conforming, accessed |

说完代码/数据段之后，我们来看下当S位的值为0时候，则表示当前是系统段描述符，系统段分为以下内容：

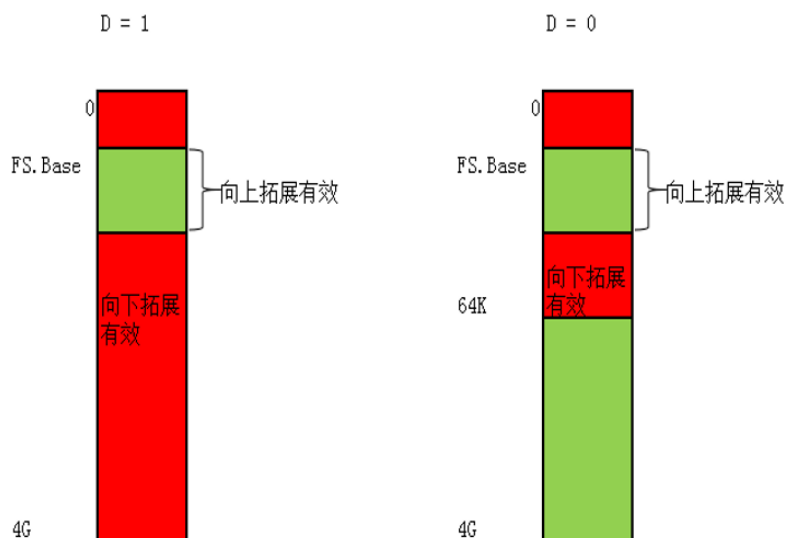
Table 3-2. System-Segment and Gate-Descriptor Types

| Type Field | | | | | Description |
|------------|----|----|---|---|------------------------|
| Decimal | 11 | 10 | 9 | 8 | |
| 0 | 0 | 0 | 0 | 0 | Reserved |
| 1 | 0 | 0 | 0 | 1 | 16-Bit TSS (Available) |
| 2 | 0 | 0 | 1 | 0 | LDT |
| 3 | 0 | 0 | 1 | 1 | 16-Bit TSS (Busy) |
| 4 | 0 | 1 | 0 | 0 | 16-Bit Call Gate |
| 5 | 0 | 1 | 0 | 1 | Task Gate |
| 6 | 0 | 1 | 1 | 0 | 16-Bit Interrupt Gate |
| 7 | 0 | 1 | 1 | 1 | 16-Bit Trap Gate |
| 8 | 1 | 0 | 0 | 0 | Reserved |
| 9 | 1 | 0 | 0 | 1 | 32-Bit TSS (Available) |
| 10 | 1 | 0 | 1 | 0 | Reserved |
| 11 | 1 | 0 | 1 | 1 | 32-Bit TSS (Busy) |
| 12 | 1 | 1 | 0 | 0 | 32-Bit Call Gate |
| 13 | 1 | 1 | 0 | 1 | Reserved |
| 14 | 1 | 1 | 1 | 0 | 32-Bit Interrupt Gate |
| 15 | 1 | 1 | 1 | 1 | 32-Bit Trap Gate |

DB位

DB位是段描述符高位4字节中的第22位，DB位的值有以下三种场景：

- 对CS段的影响：
 - DB = 0：采用32位寻址方式
 - DB = 1：采用16位寻址方式
- 对SS段的影响：
 - DB = 1：隐式栈访问指令（如：PUSH POP CALL）使用32位栈指针寄存器ESP
 - DB = 0：隐式栈访问指令（如：PUSH POP CALL）使用16位栈指针寄存器SP
 - 隐式栈访问指令：例如PUSH EBP，这句指令没有出现ESP却修改了ESP的值，所以我们就可称这种指令为隐式栈访问指令
- 向下扩展的数据段：
 - DB = 1：段上线为4GB
 - DB = 0：段上线为64KB
 - 实际上是限制扩展有效范围，大致如下：

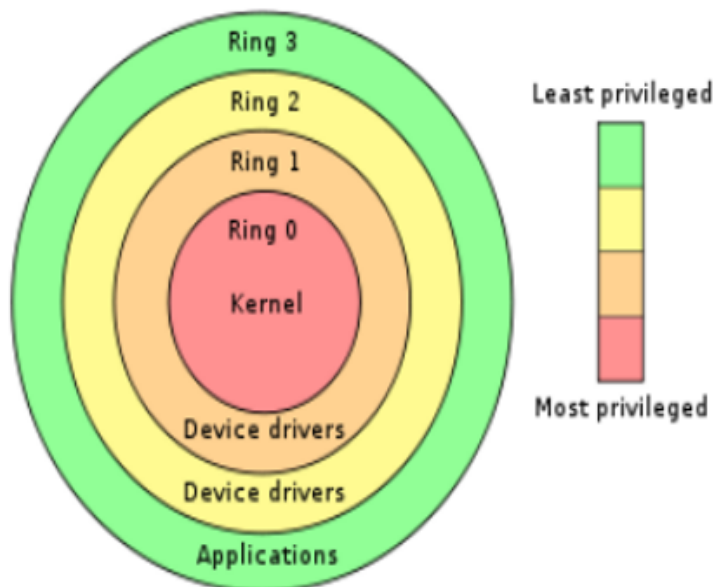


2.3 段权限

2.3.1 权限检查

我们之前所知道当向段寄存器写入时，CPU会根据段选择子去查表，并将对应内容进行检查并填入到段寄存器中，这里面就涉及到段权限检查。

在了解段权限检查之前我们先来看一下CPU的分级，如下图所示就是CPU权限的不同分级：

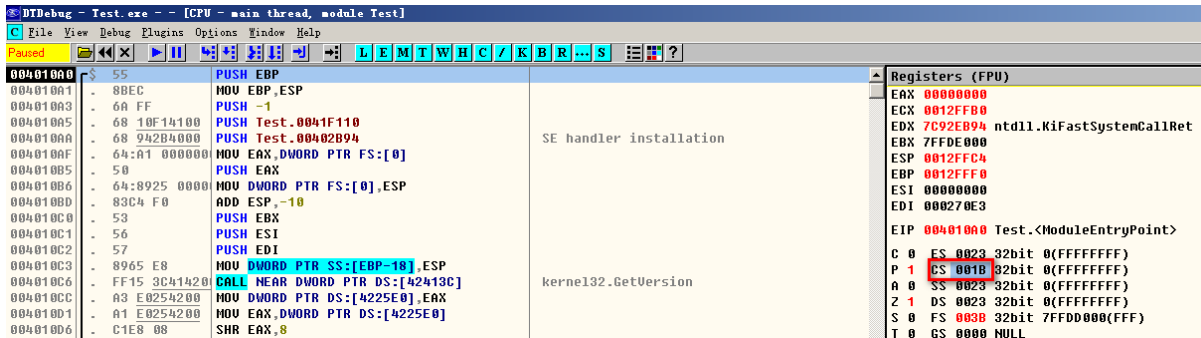


在Windows操作系统当中使用了Ring 3（应用层）、Ring 0（内核/驱动层）的权限分级，我们也可以称之为0环、3环。

判断程序权限

我们可以通过CPL（Current Privilege Level，当前特权级）来判断程序的权限等级，这里的CPL实际上就是CS和SS中存储的段选择子的最后2位（该位与其他段寄存器段选择子的RPL位是同一位置）。

如下图所示，我将一个EXE文件丢入DTDebug中，并找到它的CS中的段选择子0x001B：



所以我们根据B = 1011得出CPL = 0011 = 3，也就表示当前进程的权限是Ring 3（应用层，3环）。

DPL

DPL（Descriptor Privilege Level，描述符特权级别）存储在段描述符中，其规定了访问该段所需要的特权级别是什么。

如下指令中段选择子AX指向的段描述符中的DPL为0，但是当前程序的CPL为3，那么该指令是无法执行成功的：

```
1 MOV DS, AX
```

RPL

RPL（Request Privilege Level，请求特权级别）是针对段选择子而言的，每个段的选子都有自己的RPL。

如下指令中段选择子指向的是同一个段描述符，但是它们的RPL不一样：

```
1 MOV AX, 0x0008
2 MOV DS, AX
3
4 MOV AX, 0x000B
5 MOV DS, AX
```

数据段的权限检查

在前面的学习中我们了解了CPL、DPL、RPL，那么它们之间在数据段中又是如何进行检查的呢？如下代码（当前程序处于0环，也就表示CPL=0）：

```
1 MOV AX, 0x000B // 0xB = 1011, RPL = 3
```

```
2    MOV DS,AX // AX指向的段描述符的DPL = 0
```

在数据段的权限检查中是按照如下公式进行的：

```
1    // 数值上的比较
2    CPL <= DPL && RPL <= DPL
```

根据这段公式，我们的RPL值并不小于等于DPL，因此我们所执行的向段寄存器写值指令是无法执行成功的。

注意：代码段和系统段描述符中的检查方式是不一样的，这些将于后面的章节中了解到。

为什么需要有RPL

我们有了CPL、DPL实际上已经完成了最基本的段权限检查，为什么还需要RPL呢？我们以文件读写为例，我们可以对一个文件进行读写，但是为了避免出错，在大多情况下，如果你只需要读取权限，那么你默认会使用只读模式去打开文件。因此，在这里也是同样的道理，**你有高权限并不代表你一定需要使用高权限，为了稳定性、安全性，以最小权限原则满足对应场景即可。**

2.3.2 代码跨段

代码跨段本质上就是修改CS段寄存器，我们之前有了解到读写段寄存器可以使用MOV或LXX指令进行，但是由于CS段寄存器的特殊性我们没办法之间修改。

CS是代码段，它的改变就意味着EIP的改变，修改CS的同时就必须修改EIP，所以微软也没有直接提供以上所述的类似指令来修改CS段寄存器。那么我们想同时修改CS与EIP，可以使用如下这些指令：

```
1    JMP FAR
2    CALL FAR
3    RETF
4    INT
5    IRETD
```

本章节主要了解一下JMP FAR指令是如何执行的。

流程

JMP FAR指令，你可以称之为长跳转，如下就是JMP FAR指令格式：

```
1    // JMP FAR 段选择子:要跳转的地址
2    JMP FAR 0x20:0x004183D7
```

那么CPU遇到这样的指令它的流程是怎么样的呢？它一共有5个步骤，我们以如上指令带入看一下：

1. **将段选择子拆分：**0x20 = 0000 0000 0010 0000, RPL = 00, TI = 0, Index = 0100(十进制：4)；
2. **根据段选择子查表获取段描述符：**因为TI=0，所以查GDT表，又根据Index=4找到对应的段描述符，如果段描述符为代码段、系统段（调用门、TSS任务段、任务门），那么该指令是允许跳转的；
3. **权限检查：**根据代码段是否一致来做权限检查（XPL数值比较），如果是非一致代码段则要求CPL==DPL&&RPL<=DPL，如果是一致代码段则要求CPL>=DPL；

- a. **非一致代码段**：我们又称之为**普通代码段**，它只允许同级访问，例如Ring3只能访问Ring3（Ring0同理可得），严禁不同级别的访问；
- b. **一致代码段**：我们又称之为**共享段**，特权级高的程序不允许访问特权级低的数据，Ring0不允许访问Ring3的数据，特权级低的程序可以访问到特权级高的数据，但特权级不会发生改变，Ring3还是Ring3。
4. **加载段描述符**：通过权限检查之后，CPU会将段描述符加载到CS段寄存器中；
5. **代码执行**：CPU将CS.Base + Offset的值写入到EIP，然后执行CS:EIP处的代码，段间跳转即结束。

直接对代码段进行JMP或者CALL的操作，无论目标是一致代码段还是非一致代码段，CPL都不会发生改变。如果要提升CPL的权限，只能通过调用门或类似方法。

实验

之前我们已经了解了JMP FAR指令的执行流程了，我们可以来实际实验一下。

非一致代码段描述符

首先通过Windbg找到一个非一致的代码段描述符（通过段描述符的S位和Type位来确定），然后复制一份写入到GDT表中。

如下图所示标记出的内容，就是一个非一致的代码段描述符（S位=1 && Type域中的C位=0）：

```
kd> r gdtr
gdtr=8003f000
kd> dq 8003f000
ReadVirtual: 8003f000 not properly sign extended
8003f000  00000000`00000000 00cf9b00`0000ffff
8003f010  00cf9300`0000ffff 00cf9b00`0000ffff
8003f020  00cff300`0000ffff 80008b04`200020ab
8003f030  ffc093df`f0000001 0040f300`0000ffff
8003f040  0000f200`0400ffff 00000000`00000000
8003f050  80008955`97000068 80008955`97680068
8003f060  00009302`2f30ffff 0000920b`80003fff
8003f070  ff0092ff`700003ff 80009a40`0000ffff
```

我们使用如下指令将标记的段描述符放入GDT表中的空白部分（因为如果我们直接使用现有的值去测试，看起来不够直观）：

```
1  eq 8003f048 00cf9b00`0000ffff
```

那么接着我们需要构建一个JMP FAR指令，首先是段选择子部分，我们要让CPU根据段选择子的Index位找到8003f048这个地址的段描述符，又知道CPU根据Index位的值乘以8再加上GDT的基地址，以及满足权限检查要求CPL==DPL&&RPL<=DPL，因此段选择子为：0000 0000 0100 1011。最终指令如下：

```
1  JMP FAR 4B:004010CC
```

我们可以在DTDebug中写入这段汇编指令，然后F7一下你就会发现这时候程序成功跳转到0x004010CC位置，并且EIP和CS段寄存器的值都发生了改变：


```

004010A0  EA CC104000 JMP FAR 004B:004010CC
004010A7  90 NOP
004010A8  90 NOP
004010A9  90 NOP
004010AA  68 942B4000 PUSH Test.00402B94
004010AF  64:A1 000000 MOV EAX,DWORD PTR FS:[0]
004010B5  50 PUSH EAX
004010B6  64:8925 0000 MOV DWORD PTR FS:[0],ESP
004010BD  83C4 F0 ADD ESP,-10
004010C0  53 PUSH EBX
004010C1  56 PUSH ESI
004010C2  57 PUSH EDI
004010C3  8965 E8 MOV DWORD PTR SS:[EBP-18],ESP
004010C6  FF15 3C41420 CALL NEAR DWORD PTR DS:[42413C]
004010CC  A3 E0254200 MOV DWORD PTR DS:[4225E0],EAX
  
```

Registers (FPU)

| | |
|-----|------------------------------------|
| EAX | 00000000 |
| ECX | 0012FFB0 |
| EDX | 7C92EB94 ntdll.KiFastSystemCallRet |
| EBX | 7FFDA000 |
| ESP | 0012FFC4 |
| EBP | 0012FFF0 |
| ESI | 00000000 |
| EDI | 00010F43 |
| EIP | 004010CC Test.004010CC |
| C 0 | ES 0023 32bit 0(FFFFFFFF) |
| P 1 | CS 004B 16bit 0(0) |
| A 0 | SS 0023 32bit 0(FFFFFFFF) |
| Z 1 | DS 0023 32bit 0(FFFFFFFF) |

所以我们就完成长跳转实验，接着我们可以来验证一下权限检查部分是否真的按流程所说那样，需要满足 $CPL=DPL \& \& RPL \leq DPL$ ，可以将之前写入的段寄存器值修改一下（修改了DPL位的值）：

```
1 eq 8003f048 00cf9b00`0000ffff
```

并且我们继续使用上面的汇编指令去运行就会发现直接出错了：

```

7C92EAF0  8B1C24 MOV EBX,DWORD PTR SS:[ESP]
7C92EAF3  51 PUSH ECX
7C92EAF4  53 PUSH EBX
  
```

至此我们也就验证了此处的权限检查逻辑。

一致代码段描述符

我们可以修改段描述符，将其变为一致代码段描述符，也就是修改Type域中的C位=1：

```
1 eq 8003f048 00cf9f00`0000ffff
```

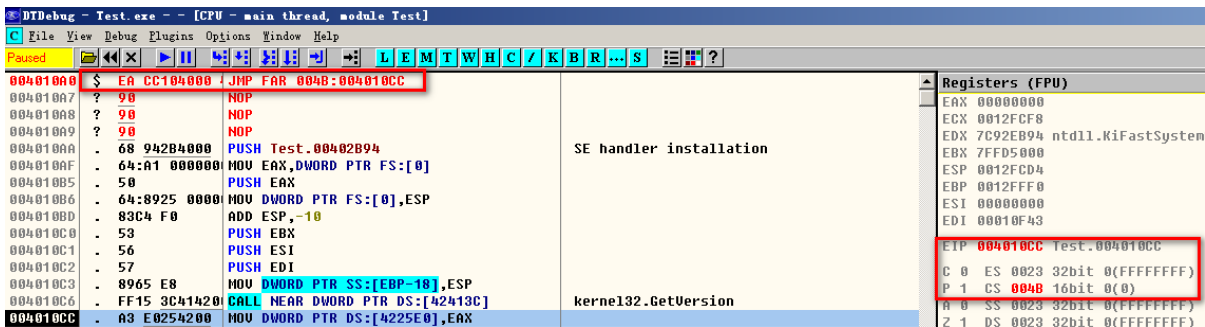
```

kd> dq 8003f000
ReadVirtual: 8003f000 not properly sign extended
8003f000  00000000`00000000 00cf9b00`0000ffff
8003f010  00cf9300`0000ffff 00cf9b00`0000ffff
8003f020  00cff300`0000ffff 80008b04`200020ab
8003f030  ffc093df`f0000001 0040f300`0000ffff
8003f040  0000f200`0400ffff 00cf9f00`0000ffff
8003f050  80008955`97000068 80008955`97680068
8003f060  00009302`2f30ffff 0000920b`80003fff
8003f070  ff0092ff`700003ff 80009a40`0000ffff
  
```

接着我们使用同样的汇编指令进行执行：

```
1 JMP FAR 4B:004010CC
```

而这里是执行成功的，这是因为一致代码段描述符的权限校验是允许低权限程序访问高权限数据的，也就是 $CPL \geq DPL$ ：



总结

- 1. 为了对数据进行保护，普通代码段是禁止不同级别进行访问的。用户态的代码不能访问内核的数据，同样，内核态的代码也不能访问用户态的数据；
- 2. 如果想提供一些通用的功能，而且这些功能并不会破坏内核数据，那么可以选择一致代码段，这样低级别的程序可以在不提升CPL权限等级的情况下访问。
- 3. 如果想访问普通代码段，只有通过调用门等提升CPL权限，才能访问。

长调用与短调用

我们可以通过JMP FAR指令实现段间的跳转，如果要实现跨段的调用就必须学习CALL FAR指令，也就是长调用。CALL FAR比JMP FAR要复杂，JMP并不影响栈，但CALL会影响。

短调用

短调用就是我们之前所学习的CALL指令，指令格式如下：

| | |
|---|-----------------|
| 1 | CALL 立即数/寄存器/内存 |
|---|-----------------|

当你执行CALL指令时，它首先会向栈中压入当前CALL指令的下一行地址，这个地址也称之为返回地址。

CALL指令执行会影响2个寄存器，一个是ESP的值（压入地址到栈，栈顶提升），一个是EIP的值（要跳转到某个地址继续执行）。

所以CALL指令也可也分解成这2个指令：

| | |
|---|-------------------|
| 1 | PUSH CALL指令的下一行地址 |
| 2 | JMP 立即数/寄存器/内存 |

我们也知道在汇编中的CALL指令大多用于调用函数的，而在每个函数调用完之后都会有一个RET指令，它的作用就是跳转回CALL指令的下一行地址，并POP。

| | |
|---|------------------|
| 1 | POP CALL指令的下一行地址 |
| 2 | JMP CALL指令的下一行地址 |

那么短调用的栈图变化就可以使用如下图来表示：



所以短调用影响的寄存器是：ESP、EIP

长调用

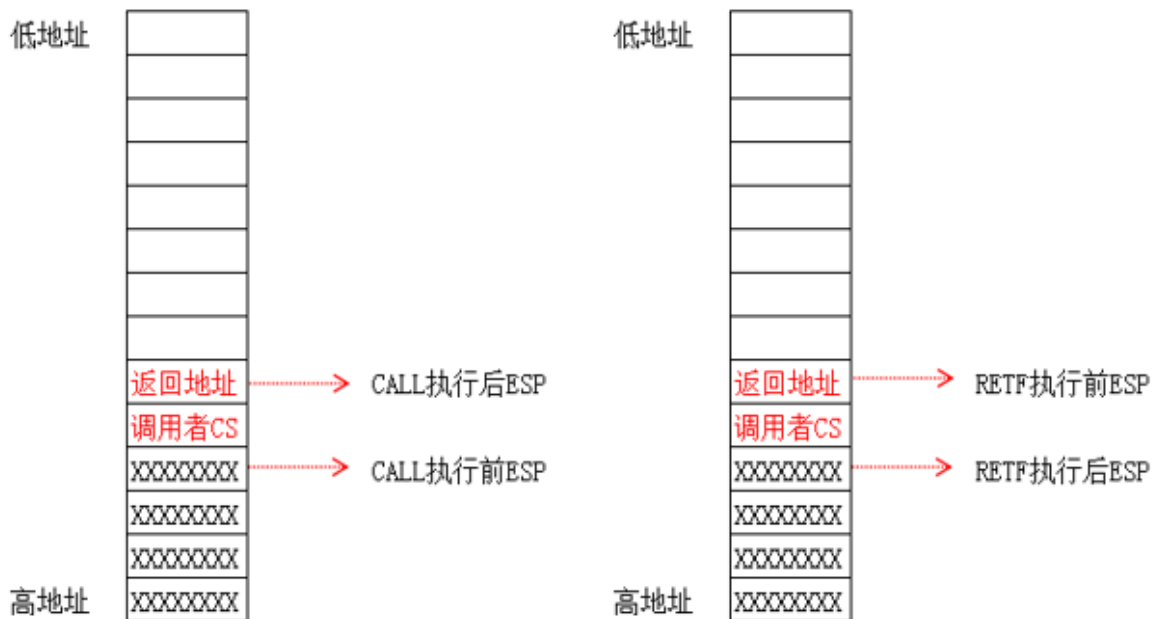
垮段不提权

长调用就是CALL FAR指令，指令格式如下：

```
1 CALL FAR CS:EIP
```

长调用分为两种情况，我们这里所讲解的长调用是指垮段不提权的。这段指令去执行的时候，地址并不会跳转到EIP，因为这里的EIP是废弃的；实际指令是根据CS段选择子查GDT表，找到表中的段描述符（这个段描述符必须是一个调用门），再根据调用门的符号计算出要调转的地址，最终再跳转过去。

这里我们所说的垮段不提权的，是指当前你的段CPL与要跳转过去的段CPL是同级的。执行长调用时，首先会压入调用者的CS，然后再压入长调用指令的下一行地址。调用完成之后，由于压入栈的内容与短调用不一样了，所以当调用完成之后不可以再使用RET来返回了，而是需要使用RETF指令，这样就可以同时恢复CS、EIP。



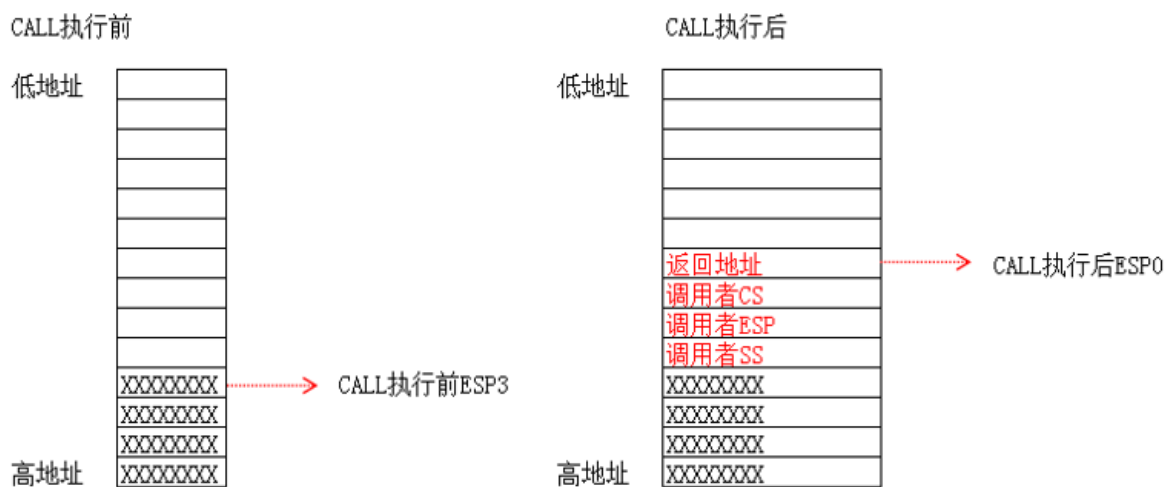
所以跨段不提权的长调用影响的寄存器就是：ESP、EIP、CS

垮段并提权

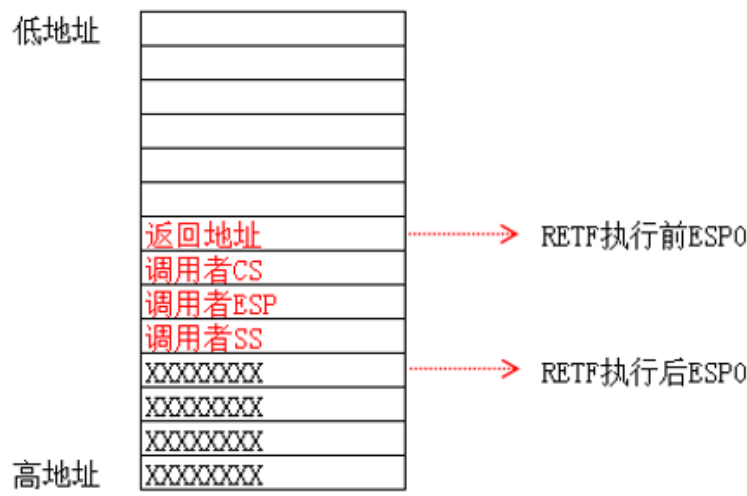
长调用的第二种情况就是跨段并提权，其与第一种情况的指令是一样的。

```
1 CALL FAR CS:EIP
```

跨段并提权的长调用指令，表示当前执行长调用指令时CPL为3，我们要去调用的段CPL为0，所以这就会产生一个提权操作，提权会改变CS的CPL，并且根据Intel的规定，**CS和SS的CPL要保持一致**，所以此时SS的值也会发生改变；除此之外，因为发生了提权，栈从3环的栈变为了0环的栈，因此ESP也会发生改变。所以为了保证我们长调用完成之后还可以恢复，就将SS、ESP、CS、返回地址压入栈中：



那么在调用结束之后的栈变化就如下图所示：



所以跨段并提权的长调用影响的寄存器就是：SS、ESP、CS、EIP

总结

- 1. 跨段调用时，一旦有权限切换，就会切换栈；
- 2. CS的权限一旦改变，SS的权限也要随着改变，CS与SS的等级必须一样；
- 3. JMP FAR只能跳转到同级非一致代码段，但CALL FAR可以通过调用门提权，提升CPL的权限，这样就可以调用非同级段。

因此接下来我们需要了解“门”相关的内容。

2.3.3 门

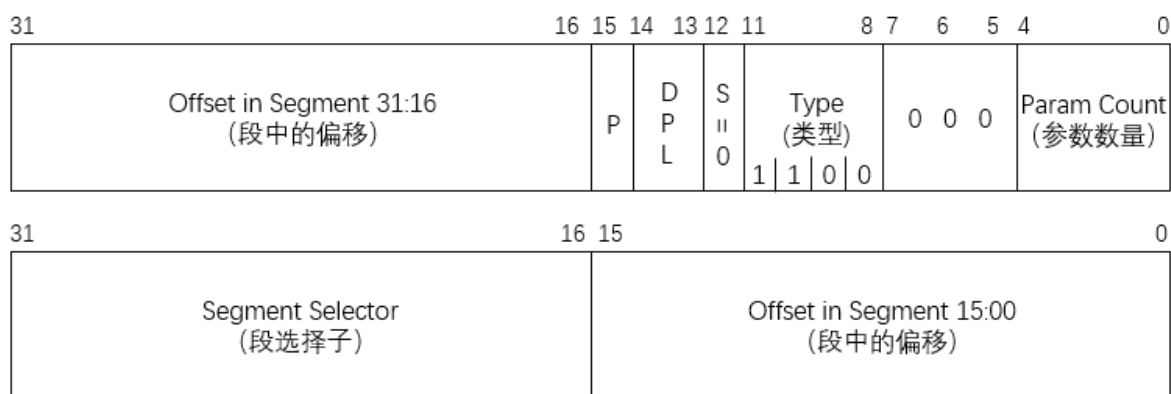
调用门

无论你是跨段提权或不提权的长调用指令，本质上都是要通过调用门去进行的，长调用指令执行流程：

```
1 CALL FAR CS:EIP
```

- 1. 根据CS的值查GDT表，找到对应的段描述符，这个描述符是一个调用门描述符；
- 2. 在调用门描述符中找到它存储的另外一个代码段的段选择子；
- 3. 找到段选择子指向的段，真正要执行的地址就是指向的段,Base + 调用门描述符中的偏移地址。

调用门描述符的结构如下图所示，我们可以看到它跟段描述符的结构有一些差别，当S位为0则表示这是一个系统段描述符，并且当Type域为1100时则确定这是一个调用门描述符：



在低32位中的第16到31位就是段选择子，在低32位中第0到第15位及高32位中过的第16到31位存储的就是偏移。

构造调用门

由于Windows中没有使用调用门，所以我们需要自己去构建一个调用门，也就是向GDT表中写一个调用门描述符。

我们按结构图的方式从高位开始构建调用门描述符，首先我们不确定要去的地方是哪，所以偏移31:16先填上0，接着我们之前知道P位必须为1才表示这是一个有效的段描述符，DPL必须为3因为我们从3环使用长调用指令，S位必须为0，Type域为1100，后面的第5、6、7位默认为0，第0到第4位这里写0即可，因为我们当前不需要传参，所以我们得出高32位为：

```

1  0000 0000 0000 0000 -> Offset
2  1110 -> P、DPL、S
3  1100 -> Type
4  0000 0000 -> 0 - 7位
5
6  0x0000EC00

```

低32位段选择子可以设为0x0008（对应0环代码段）、0x001B（对应3环代码段），偏移不确定所以填0：

```

1  0x00080000

```

最终我们得出调用门描述符为：

```

1  0000EC00`00080000

```

我们可以在Windbg中使用eq指令覆盖GDT表中没有被用到的段描述符（都为0）：

```

1  eq 8003f048 0000EC00`00080000

```

```

kd> r gdtr
gdtr=8003f000
kd> dq 8003f000
ReadVirtual: 8003f000 not properly sign extended
8003f000  00000000`00000000 00cf9b00`0000ffff
8003f010  00cf9300`0000ffff 00cffb00`0000ffff
8003f020  00cff300`0000ffff 80008b04`200020ab
8003f030  ffc093df`f0000001 0040f300`00000fff
8003f040  0000f200`0400ffff 00000000`00000000
8003f050  80008955`97000068 80008955`97680068
8003f060  00009302`2f30ffff 0000920b`80003fff
8003f070  ff0092ff`700003ff 80009a40`0000ffff
kd> eq 8003f048 0000EC00`00080000
WriteVirtual: 8003f048 not properly sign extended
kd> dq 8003f000
ReadVirtual: 8003f000 not properly sign extended
8003f000  00000000`00000000 00cf9b00`0000ffff
8003f010  00cf9300`0000ffff 00cffb00`0000ffff
8003f020  00cff300`0000ffff 80008b04`200020ab
8003f030  ffc093df`f0000001 0040f300`00000fff
8003f040  0000f200`0400ffff 0000ec00`00080000
8003f050  80008955`97000068 80008955`97680068
8003f060  00009302`2f30ffff 0000920b`80003fff
8003f070  ff0092ff`700003ff 80009a40`0000ffff

```

这样我们就构建好调用门了。

调用门使用

无参数

我们知道如何构建调用门了，接下来我们使用调用门，如下代码所示，我们使用长调用指令，给的段选择子为0x48，这是因为它拆分出来对应的GDT表的描述符正好是我们上面写入的调用门描述符，由于EIP是废弃没用过的，所以我们随便赋值：

```

1  #include <windows.h>
2  #include <stdio.h>
3
4  void __declspec(naked) GetRegister() {
5      _asm {
6          int 3
7          retf
8      }
9  }
10
11 void main()
12 {
13     char buff[6];
14     *(DWORD*)&buff[0] = 0x12345678; // EIP, 废弃
15     *(WORD*)&buff[4] = 0x48; // 段选择子
16     _asm {

```

```

17         call far fword ptr[buf]
18     }
19     getchar();
20 }

```

而我们之前构建的调用门描述符中的选择子是指向0环的代码段，在Windows里所有数据、代码段的Base都为0，所以我们真正跳转的位置就是偏移位指向的内容。因此，我们要运行如上这段代码的话一定是报错的，因为我们给的偏移位置是0，我们预期效果是想要长跳转到GetRegister函数，所以我们可以下断点先将该函数的地址获取下来。

如下图所示，我们获得GetRegister函数的函数地址0x0040B4B0：

```

1:  #include <windows.h>
2:  #include <stdio.h>
3:
4:  void __declspec(naked) GetRegister() {
0040B4B0    int     3
5:      _asm {
6:          int 3
7:          retf
0040B4B1    retf

```

所以我们可以将调用门描述符按格式重新写入GDT表中：

| | |
|---|-------------------|
| 1 | 0040EC00`0008B4B0 |
|---|-------------------|

```

kd> eq 8003f048 0040EC00`0008B4B0
WriteVirtual: 8003f048 not properly sign extended
kd> dq 8003f000
ReadVirtual: 8003f000 not properly sign extended
8003f000  00000000`00000000 00cf9b00`0000ffff
8003f010  00cf9300`0000ffff 00cffb00`0000ffff
8003f020  00cff300`0000ffff 80008b04`200020ab
8003f030  ffc093df`f0000001 0040f300`00000fff
8003f040  0000f200`0400ffff 0040ec00`0008b4b0
8003f050  80008955`97000068 80008955`97680068
8003f060  00009302`2f30ffff 0000920b`80003fff
8003f070  ff0092ff`700003ff 80009a40`0000ffff

```

这样我们运行代码时候长跳转到GetRegister函数执行，该函数内有一个INT 3，这是用来断点的（跟软件调试有关，原理在后续章节中讲解），所以代码执行到该函数时候就会中断；并且我们需要注意虽然当前代码是在低2G中，但是通过调用门这段代码的权限就已经提升为0环的权限了，也因此，我们在3环是没法捕捉到这个断点的，只能在0环的调试器中去捕捉，也就是中断到我们的Windbg调试器。

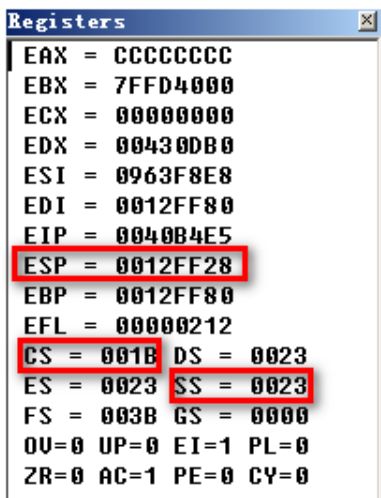
我们执行这段代码会发现，Windbg确实有反应，并且提示我们在这个地址中断了：

```

kd> g
Break instruction exception - code 80000003 (first chance)
0040b4b0 cc          int     3

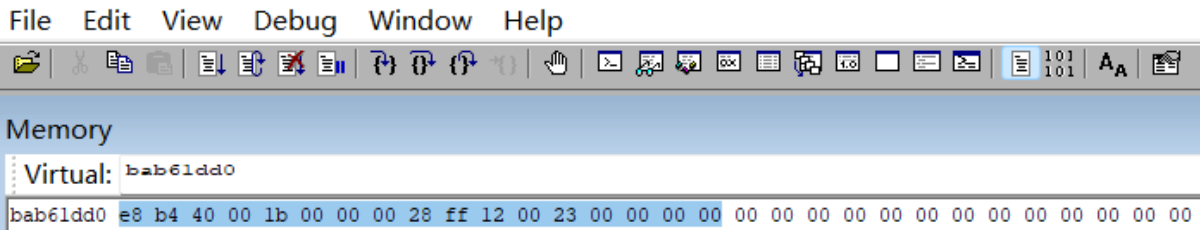
```

我们也可以对比下长调用前后的寄存器信息，发现SS、ESP、CS寄存器都发生了变化：



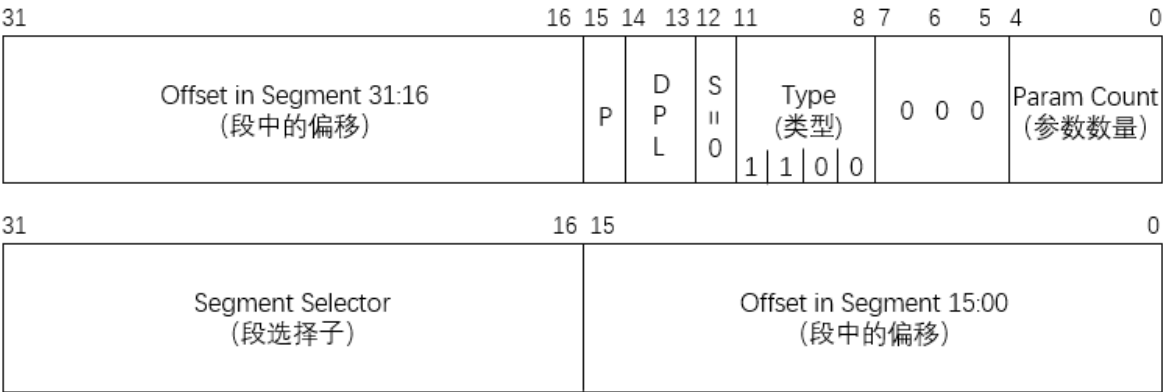
```
kd> r
eax=cccccccc ebx=7ffde000 ecx=00000000 edx=00430db0 esi=0993f8e8 edi=0012ff80
eip=0040b4b0 esp=blcf9dd0 ebp=0012ff80 iopl=0         nv up ei pl nz ac po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000212
0040b4b0 cc                      int     3
```

并且我们观察栈的内容，会发现确实是如我们之前所看的长调用权限提升的栈图一样，将SS、ESP、CS、返回地址压入栈中：



有参数

之前我们使用调用门时没有传递参数，但实际上调用门是支持参数传递的，所以我们来看下如何实现。根据调用门结构我们知道它的高32位第0位至4位表示的就是参数数量，也就是你要传递的参数的个数：



所以我们可以给之前构建的调用门描述符添加上参数个数3：

```
1 0040EC03`0008B4B0
```

接下来我们可以使用如下代码去使用参数，通过push压入参数，调用时根据ESP寻址获取参数，有个细节需要注意这里由于我们压入了3个参数，导致栈发生了变化，所以使用RETF指令时候需要加上0xC，这样执行完代码后才能保持栈平衡：

```
1  #include <windows.h>
2  #include <stdio.h>
3
4  int a,b,c = 0;
5  int o_eax = 0;
6
7  void __declspec(naked) GetParam() {
8      _asm {
9          mov o_eax, eax
10         mov eax, [esp+8]
11         mov c, eax
12         mov eax, [esp+0xC]
13         mov b, eax
14         mov eax, [esp+0x10]
15         mov a, eax
16         mov eax, o_eax
17
18         retf 0xC
19     }
20 }
21
22 void main()
23 {
24
25     char buff[6] = {0};
26     *(DWORD*)&buff[0] = 0x12345678;
27     *(WORD*)&buff[4] = 0x48;
28
29     _asm {
30         push 1
31         push 2
32         push 3
33         call far fword ptr[buff]
34     }
35
36     printf("%d %d %d", a, b, c);
37
38     getchar();
39 }
```

我们执行这段程序发现确实可以通过这种方式获取到传递的参数值：

```

int a,b,c = 0;
int o_eax = 0;

void __declspec(naked) GetParam() {
    _asm {
        mov o_eax, eax
        mov eax, [esp+8]
        mov c, eax
        mov eax, [esp+0xc]
        mov b, eax
        mov eax, [esp+0x10]
        mov a, eax
        mov eax, o_eax

        retf 0xc
    }
}

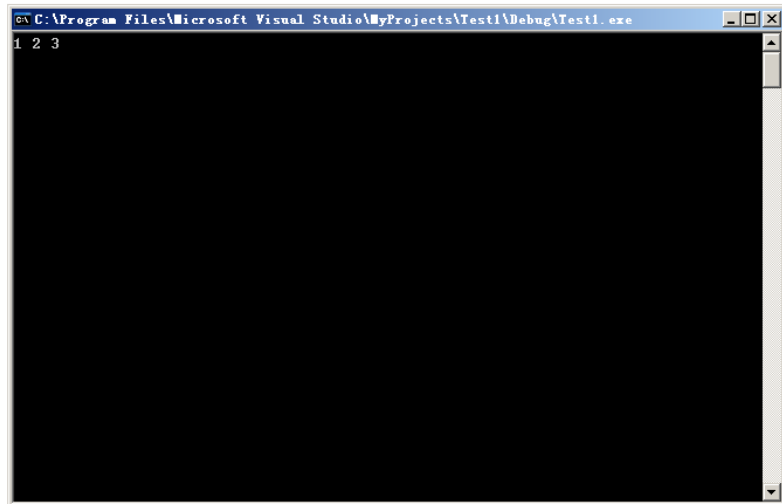
void main()
{
    char buff[6] = {0};
    *(DWORD*)&buff[0] = 0x12345678;
    *(WORD*)&buff[4] = 0x48;

    _asm {
        push 1
        push 2
        push 3
        call far fword ptr[buff]
    }

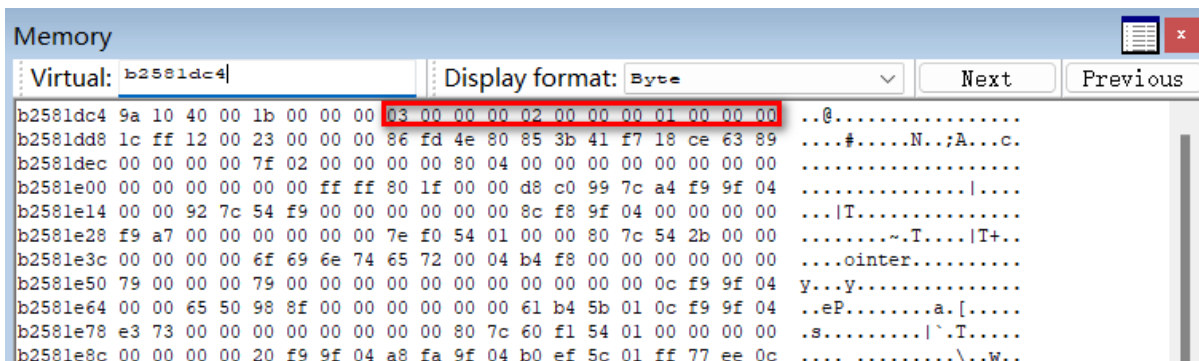
    printf("%d %d %d", a, b, c);

    getch();
}

```



那么为什么代码中使用ESP寻址是这个对应的偏移量呢，我们需要在汇编中写入一个INT 3指令让程序中断在Windbg中，观察一下栈。



根据栈我们知道我们传递的参数是夹在调用者ESP和调用者CS中间的，也就是如下图展示的这样：



所以在代码中我们就可以分别使用0x8、0xC、0x10偏移量去将参数取出来。

总结

- 1. 当通过门，权限不变时，只会PUSH两个值：CS和返回地址，新的CS的值由调用门决定；
- 2. 当通过门，权限改变的时候，会PUSH四个值：SS，ESP，CS，返回地址，新的CS由调用门决定，新的SS和ESP由TSS提供（后面会讲TSS）；
- 3. 通过调用门时，由调用门决定要执行的代码，但使用RETF返回时，由栈中压入的值决定，也就是说，进门时只能按指定路线走，出门时可以翻墙（只要改变栈里面的值就可以想去哪去哪）；
- 4. 可不可以再建个门出去呢？也就是用CALL指令，当然可以，前门进，后门出。

中断门

Windows没有使用调用门，但是使用了中断门，当我们有了调用门的基础之后，中断门的学习就很轻松了。

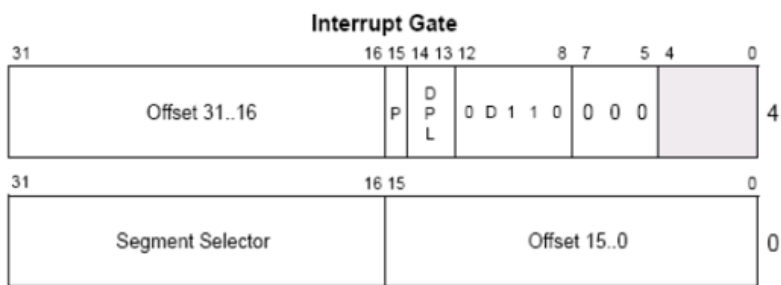
中断门有两个场景：**1.系统调用**：大家在开发应用程序时都会使用到Windows提供的API，这些API在执行的过程中需要从3环一步一步进入到0环，这一过程就是系统调用，在这个过程中也用到了中断门（一些比较老的API使用的是中断门，但是新API中使用的都是快速调用）；**2.调试**：大家使用OD调试程序时候会使用断点，断点本质上就是在你选中的这一行中写入一个字节0xCC，也就是INT 3指令，这个指令就是用来执行中断门的。

中断门也有一张表，我们称之为IDT（中断描述符表），与GDT一样，IDT也是由一系列描述符组成的，每个描述符占8字节，需要注意的是IDT表中的第一个元素不是NULL（GDT是）。

与GDT一样，我们也可以在Windbg中查看IDT表的基址和长度：

| | |
|---|--------|
| 1 | r idtr |
| 2 | r idtl |

在IDT表中有3种门描述符：1.任务门描述符；2.中断门描述符；3.陷阱门描述符。我们来看一下中断门描述符的结构：



我们会发现中断门描述符与调用门描述符差别不是很大，其中它多了一个D位，该值为0时表示16位中断门，为1时表示32位中断门，其他的都一样。

接着我们来看一下中断门的执行流程，实际上和调用门差别不是很大，可以类比的来看：

- 执行调用门的指令：CALL CS:EIP，其中CS是段选择子，包含了查找GDT表的是一个索引；
- 执行中断门的指令：INT N，其中N是IDT表的一个索引。

执行流程就只有这个差别，当CPU通过N这个索引在IDT表中找到了中断门描述符后，执行的步骤就和调用门的步骤完全一样了，可以参考调用门的执行流程。

当找到中断门描述符后，CPU还是会通过中断门描述符里的段选择子，去GDT表中找到对应的段描述符，**段描述符的基址加上当前中断门描述符的段内偏移就是需要跳转代码段的地址**。所以说中断门的执行会查找两张表，先查找IDT表，再查找GDT表。

与调用门使用长返回RET F不同，中断门使用中断返回指令：IRET/IRETD。

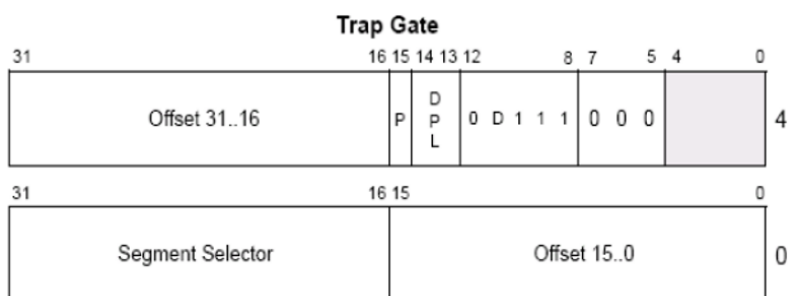
INT N指令：

- 在没有权限切换时，会向栈压入3个值，分别是CS，EFLAG，返回地址；
- 在有权限切换时，会向栈压入5个值，分别是SS，ESP，EFLAG，CS，返回地址。

这也是与调用门不同的地方，中断门会多压入一个值，这也就说明EFLAG通过中断门跨段时，值会改变。

陷阱门

陷阱门与中断门几乎是一样的，陷阱门描述符也存储在IDT表中，其与中断门描述符唯一不同的是高32位的第8位为1：



除此之外陷阱门和中断门的区别在于，中断门执行后EFLAG寄存器的值发生了改变，而陷阱门不会改变EFLAG的值。

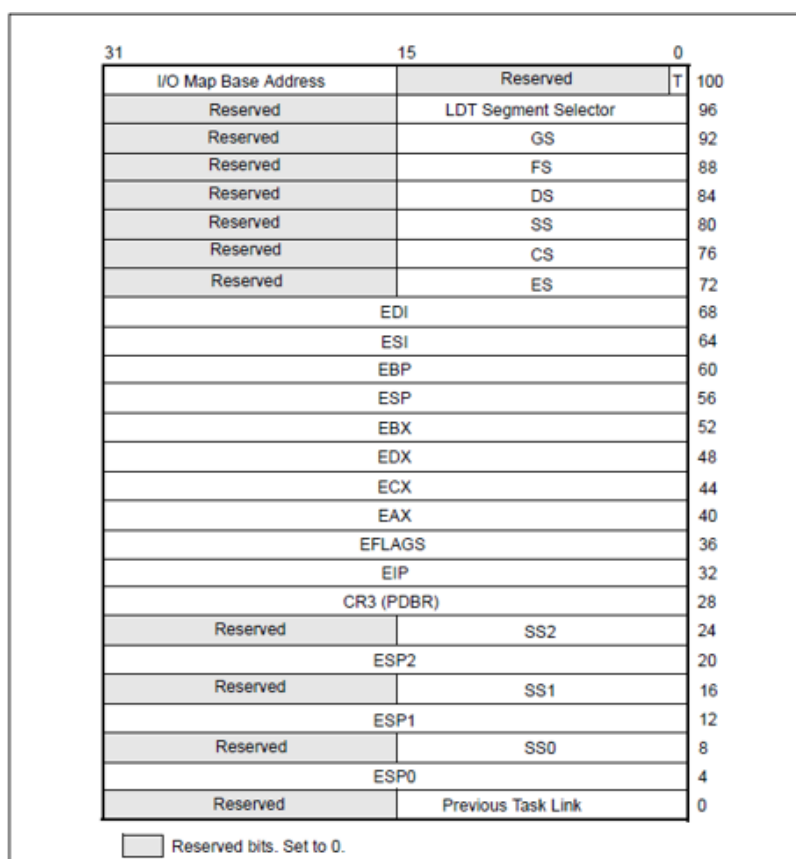
中断门会修改EFLAG寄存器中的IF位的值，IF标志是用于控制处理器对可屏蔽中断请求的响应。**置1以响应可屏蔽中断，反之则禁止可屏蔽中断**。IF标志对不可屏蔽中断没有影响。

任务段

在以上章节中我们了解到当使用调用、中断、陷阱门时，当出现权限切换的时候栈也会随之切换，并且由于CS的CPL发生了改变，也就导致SS必须进行切换。**我们知道CS的值是由门来指定的，但是ESP和SS呢？它们基于TSS（Task-state segment，任务段）得到的值。**

TSS结构

TSS本质上就是一块内存，这块内存的大小是104字节，如下图所示就是它的结构，我们可以看见TSS里包含了所有寄存器的值：



TSS作用

Intel的设计思想就是当你要去创建一个新的任务（线程）时应该有一个新的环境，而不是基于老的环境去做，这也就需要换掉你的所有寄存器，因此你可以从TSS中把各值拉过来给到寄存器，并且将原先的寄存器的值再放入TSS中，这样可以直接恢复原先的环境。

但这只是Intel的设计思想，操作系统并没有这样做，所以我们理解TSS的话，只需要知道它可以一次性换掉所有的寄存器即可。

那么我们也可以根据TSS的结构知道，里面的ESP0、ESP1、ESP2就表示0环、1环、2环，SS[0-2]也是如此，所以我们也就知道权限切换时ESP、SS的值是从TSS中获取的。

流程

CPU取值的流程如下图所示，根据TR寄存器（Task Register）的段选择子到GDT表中找到TSS段描述符，得到了TSS的基址、以及段大小，这样就能定位到TSS这块内存：

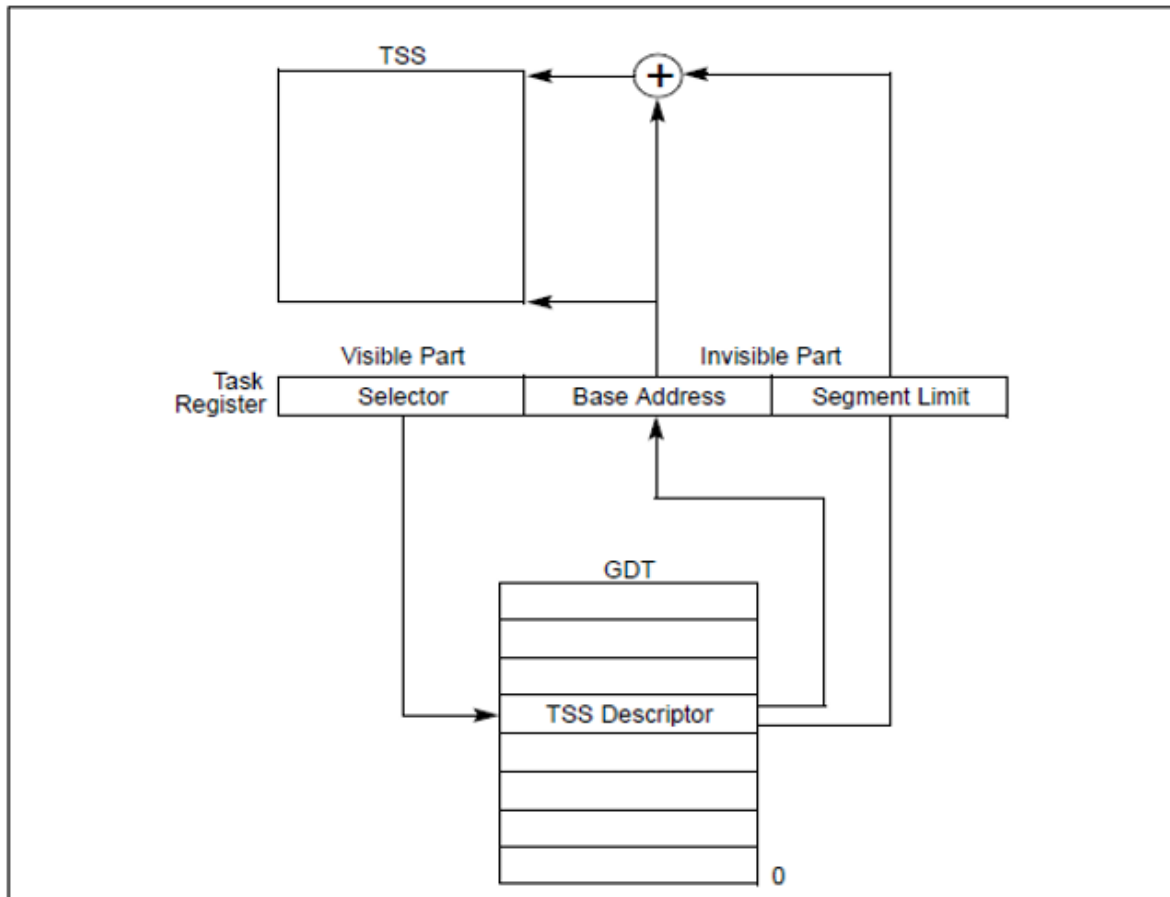


Figure 7-5. Task Register

TSS段描述符

如下图所示就是TSS段描述符的结构，TSS段描述符也属于系统段描述符的一种，所以高4字节的第12位为0，当Type位为1001或1011就表示当前是TSS段描述符，前者表示当前段描述符没有被加载到TR寄存器中，后者则反之：

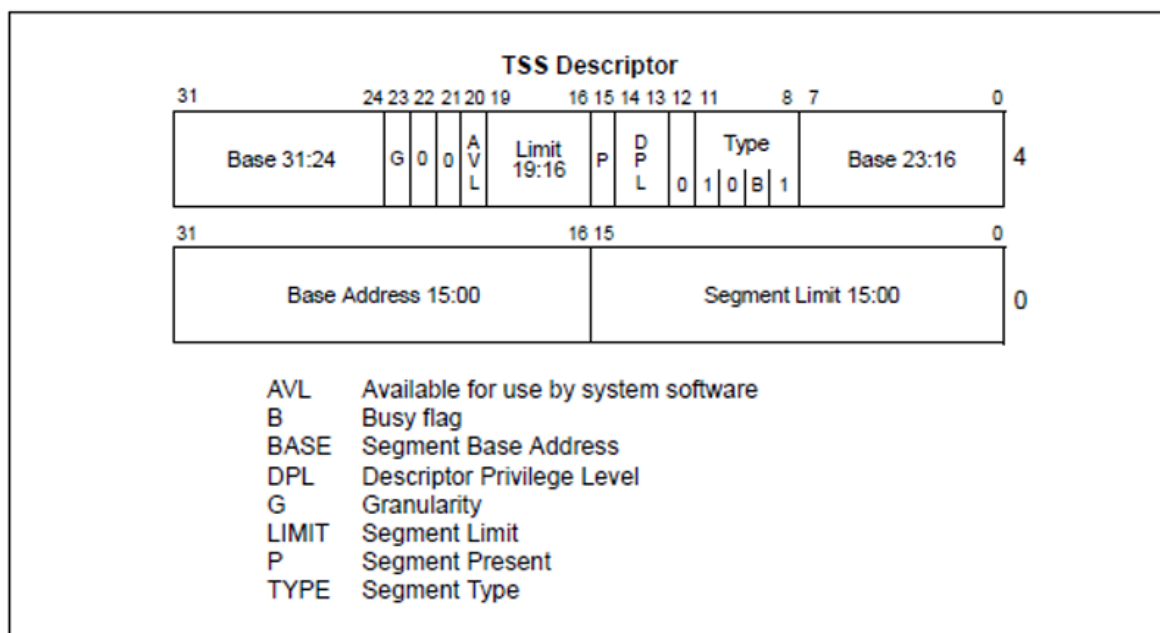


Figure 7-3. TSS Descriptor

TR段寄存器

我们想读写TR段寄存器的话不能使用MOV指令，需要使用到LTR、STR指令。

LTR指令可以将TSS段描述符加载到TR寄存器中，但只能改变TR寄存器的值（96位），并不会改变TSS；LTR指令只能在系统层使用；加载后TSS段描述符的状态位（高4字节的第9位，或可以理解为Type域从0x9/1001变成了0xB/1011）会发生变化。

STR指令可以读取TR段寄存器的内容，但是该指令只能读取TR段寄存器的16位，也就是选择子。

实现任务切换

在Windows操作系统中没有使用TSS进行任务（线程）切换，我们可以手动实现一下。主要分为几个步骤：

1. 构造完整的TSS；
2. 构造TSS段描述符；
3. 使用CALL FAR/JMP FAR指令修改TR寄存器。

构造完整的TSS

如下我们使用代码构造了一个完整的TSS：

```

1  DWORD tss[0x68] = {
2      0x00000000,    // Previous Task Link
3      // 不同权限对应的ESP、SS，如果不涉及到权限切换所以可以将这些寄存器的值全部填0
4      (DWORD)stack, // ESP0
5      0x00000010,    // SS0
6      0x00000000,    // ESP1
7      0x00000000,    // SS1
8      0x00000000,    // ESP2

```



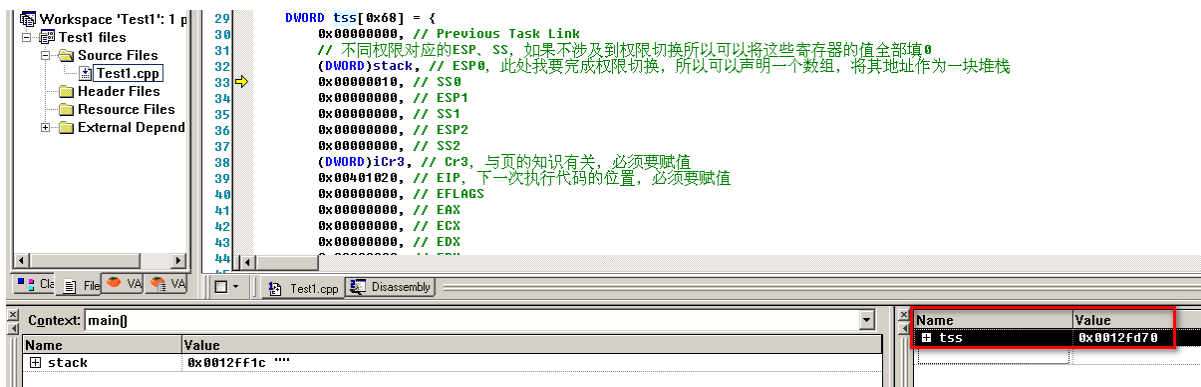
```

9      0x00000000, // SS2
10     (DWORD)iCr3, // Cr3, 与页的知识有关, 必须要赋值
11     0x00401020,   // EIP, 下一次执行代码的位置, 必须要赋值
12     0x00000000,   // EFLAGS
13     0x00000000,   // EAX
14     0x00000000,   // ECX
15     0x00000000,   // EDX
16     0x00000000,   // EBX
17     (DWORD)stack, // ESP, 任务切换时也需要切换栈, 所以在代码中我们可以声明一个数组, 将其地址作为一块栈
18     0x00000000,   // EBP
19     0x00000000,   // ESI
20     0x00000000,   // EDI
21     0x00000023,   // ES
22     0x00000008, // CS, 切到0环的代码段描述符
23     0x00000010,   // SS, CS与SS需要保持一致
24     0x00000023,   // DS
25     0x00000030,   // FS, 切到0环就是0x30, 3环就是0x3B
26     0x00000000,   // GS, Windows没有使用这个段寄存器所以永远是0
27     0x00000000,   // LDT, 填0
28     0x20ac0000 // IO_MAP, Windows2000以后不用了
29 };

```

构造TSS段描述符

根据TSS段描述符的结构构造描述符, 首先获取TSS的地址, 在代码中下断点然后查看地址即可:



其地址为0x0012fd70, 我们就得出对应TSS段描述符中Base的值, 接着Limit就是TSS的大小0x68, DPL为3 (3环程序访问), Type域为0x9即表示当前描述符没有被加载过。最终得出TSS段描述符为: **0000e912`fd700068**。

接着我们使用eq指令在Windbg中向GDT表中写入我们构造好的TSS段描述符:

```

kd> r gdtr
gdtr=8003f000
kd> dq 8003f000
ReadVirtual: 8003f000 not properly sign extended
8003f000  00000000`00000000 00cf9b00`0000ffff
8003f010  00cf9300`0000ffff 00cffb00`0000ffff
8003f020  00cff300`0000ffff 80008b04`200020ab
8003f030  ffc093df`f0000001 0040f300`00000fff
8003f040  0000f200`0400ffff 00000000`00000000
8003f050  80008955`97000068 80008955`97680068
8003f060  00009302`2f30ffff 0000920b`80003fff
8003f070  ff0092ff`700003ff 80009a40`0000ffff
kd> eq 8003f048 0000e912`fd700068
WriteVirtual: 8003f048 not properly sign extended
kd> dq 8003f000
ReadVirtual: 8003f000 not properly sign extended
8003f000  00000000`00000000 00cf9b00`0000ffff
8003f010  00cf9300`0000ffff 00cffb00`0000ffff
8003f020  00cff300`0000ffff 80008b04`200020ab
8003f030  ffc093df`f0000001 0040f300`00000fff
8003f040  0000f200`0400ffff 0000e912`fd700068
8003f050  80008955`97000068 80008955`97680068
8003f060  00009302`2f30ffff 0000920b`80003fff
8003f070  ff0092ff`700003ff 80009a40`0000ffff

```

然后我们的完整代码如下所示，：

```

1  #include <stdio.h>
2  #include <windows.h>
3
4  int o_eax, n_esp;
5  short n_cs, n_ss;
6
7  void __declspec(naked) GetValue() {
8      // 将寄存器保存到全局变量中
9      _asm {
10         mov o_eax, eax
11         mov n_esp, esp
12         mov ax, cs
13         mov n_cs, ax
14         mov ax, ss
15         mov n_ss, ax
16         mov eax, o_eax
17         iret
18     }
19 }
20
21 void main()
22 {
23     char stack[100] = {0}; // 栈
24     char buffer[6] = {0x0, 0x0, 0x0, 0x0, 0x4B, 0x0}; // 选择子
25     int iCr3 = 0;
26

```

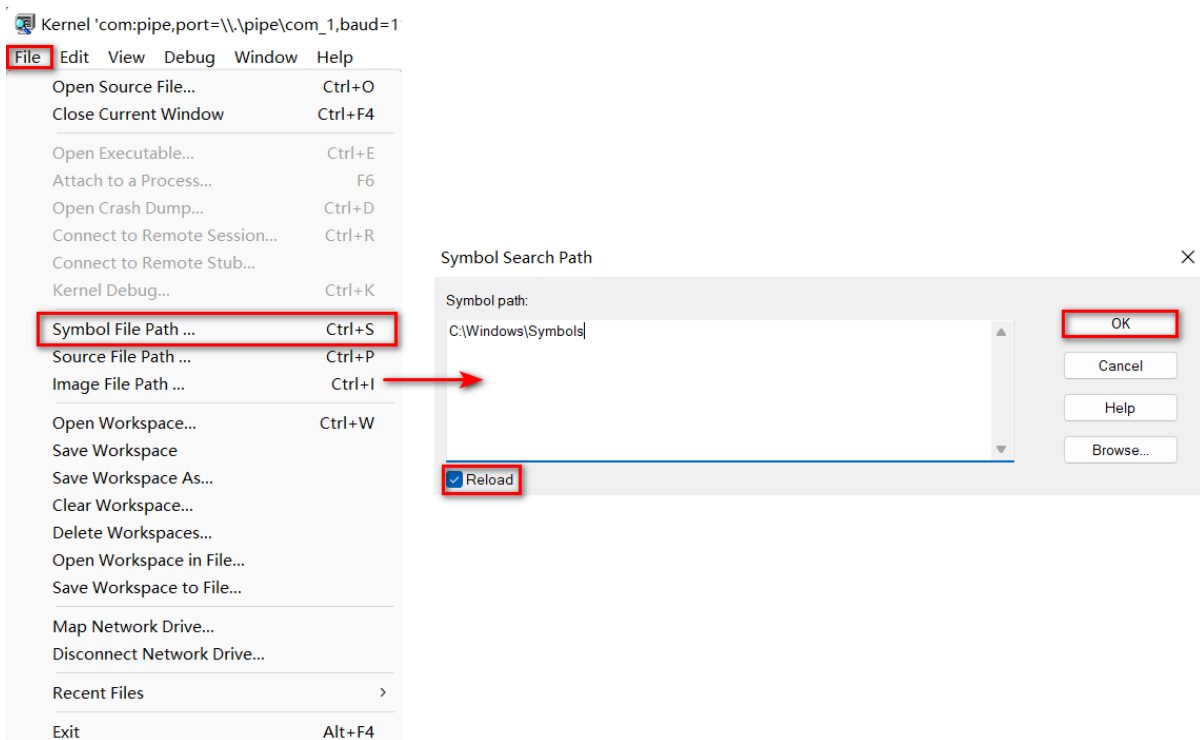
```

27     printf("Input: ");
28     scanf("%x", &iCr3);
29     getchar();
30
31     DWORD tss[0x68] = {
32         0x00000000,    // Previous Task Link
33         // 不同权限对应的ESP、SS，如果不涉及到权限切换所以可以将这些寄存器的值全部填0
34         0x00000000,    // ESP0
35         0x00000000,    // SS0
36         0x00000000,    // ESP1
37         0x00000000,    // SS1
38         0x00000000,    // ESP2
39         0x00000000,    // SS2
40         (DWORD)iCr3,    // Cr3, 与页的知识有关，必须要赋值
41         0x00401020,    // EIP, 下一次执行代码的位置，必须要赋值，在代码中就是GetValue函数的地址
42         0x00000000,    // EFLAGS
43         0x00000000,    // EAX
44         0x00000000,    // ECX
45         0x00000000,    // EDX
46         0x00000000,    // EBX
47         (DWORD)stack,    // ESP, 任务切换时也需要切换栈，所以在代码中我们可以声明一个数组，将其地址作为一块栈
48         0x00000000,    // EBP
49         0x00000000,    // ESI
50         0x00000000,    // EDI
51         0x00000023,    // ES
52         0x00000008,    // CS, 切到0环的代码段描述符
53         0x00000010,    // SS, CS与SS需要保持一致
54         0x00000023,    // DS
55         0x00000030,    // FS, 切到0环就是0x30, 3环就是0x3B
56         0x00000000,    // GS, Windows没有使用这个段寄存器所以永远是0
57         0x00000000,    // LDT, 填0
58         0x20ac0000    // IO_MAP, Windows2000以后不用了，默认值
59     };
60
61     _asm {
62         call far fword ptr[buffer] // 长调用
63     }
64
65     // 输出寄存器的值证明完成了任务切换
66     printf("ESP: %x, CS: %x, SS: %x \n", n_esp, n_cs, n_ss);
67
68     getchar();
69 }

```

接着我们需要运行程序，然后在Windbg中断点输入**!process 0 0**指令获取Cr3的值填入到程序中。但是我们需要导入符号表才能使用Windbg这个指令，所以根据我们的系统版本找到对应的离线符号表安装包（这里我是Windows XP SP2，安装包链接：<https://pan.baidu.com/s/1j0oDzGHVviMvilWgWhdoxg>，提取码：keey）。

安装完成之后接着在Windbg中按如下图所示填入符号表的目录即可加载：



接着我们按照流程来，获取到当前程序的Cr3值0x219cd000，并且输入到程序中：

```
Failed to get VadRoot  
PROCESS 894ceb88 SessionId: 0 Cid: 0958 Peb: 7ffd5000 ParentCid: 040c  
DirBase: 219cd000 ObjectTable: e237ec68 HandleCount: 15.  
Image: Test1.exe
```

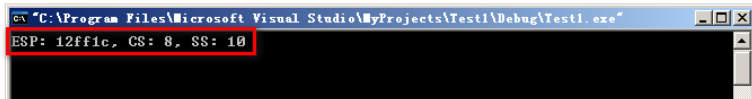


输入之后回车成功，程序输出的寄存器值就是我们TSS中赋予的，则实验成功：

```

DWORD tss[0x68] = {
    0x00000000, // Previous Task Link
    // 不同权限对应的ESP、SS，如果不涉及到权限切换所以可以将这些寄存器的值全部填0
    0x00000000, // ESP0
    0x00000000, // SS0
    0x00000000, // ESP1
    0x00000000, // SS1
    0x00000000, // ESP2
    0x00000000, // SS2
    (DWORD)iCr3, // Cr3, 与页的知识有关，必须要赋值
    0x00401020, // EIP, 下一次执行代码的位置，必须要赋值
    0x00000000, // EFLAGS
    0x00000000, // EAX
    0x00000000, // ECX
    0x00000000, // EDX
    0x00000000, // EBX
    (DWORD)stack, // ESP, 任务切换时也需要切换堆栈，所以在代码中我们可以声明一个数组，将其地址作为一块堆栈
    0x00000000, // EBP
    0x00000000, // ESI
    0x00000000, // EDI
    0x00000023, // ES
    0x00000008, // CS, 切换到0环的代码段描述符
    0x00000010, // SS, CS与SS需要保持一致
};

```



任务门

除了之前所提到的JMP/CALL指令进行任务切换外，我们还可以使用任务门，它的优势如下：

- 任务门可以放在GDT表中，也可以放在IDT表中，还能放在当前线程的LDT表中，而TSS段描述符只能在GDT表中；
- 任务门可以让低权限的线程进行任意切换，通过任务门，TSS段描述符就不再进行权限检查了（**CPL=3**的程序使用**DPL=3**的任务门，可以访问到**DPL=0**的TSS段描述符，最终可以完成任务切换）；
- 由于任务门可以位于IDT表中，所以当遇到中断或者异常时，可以切换到独立的任务去处理异常。

下面为不同表中，任务门进行任务切换的过程：

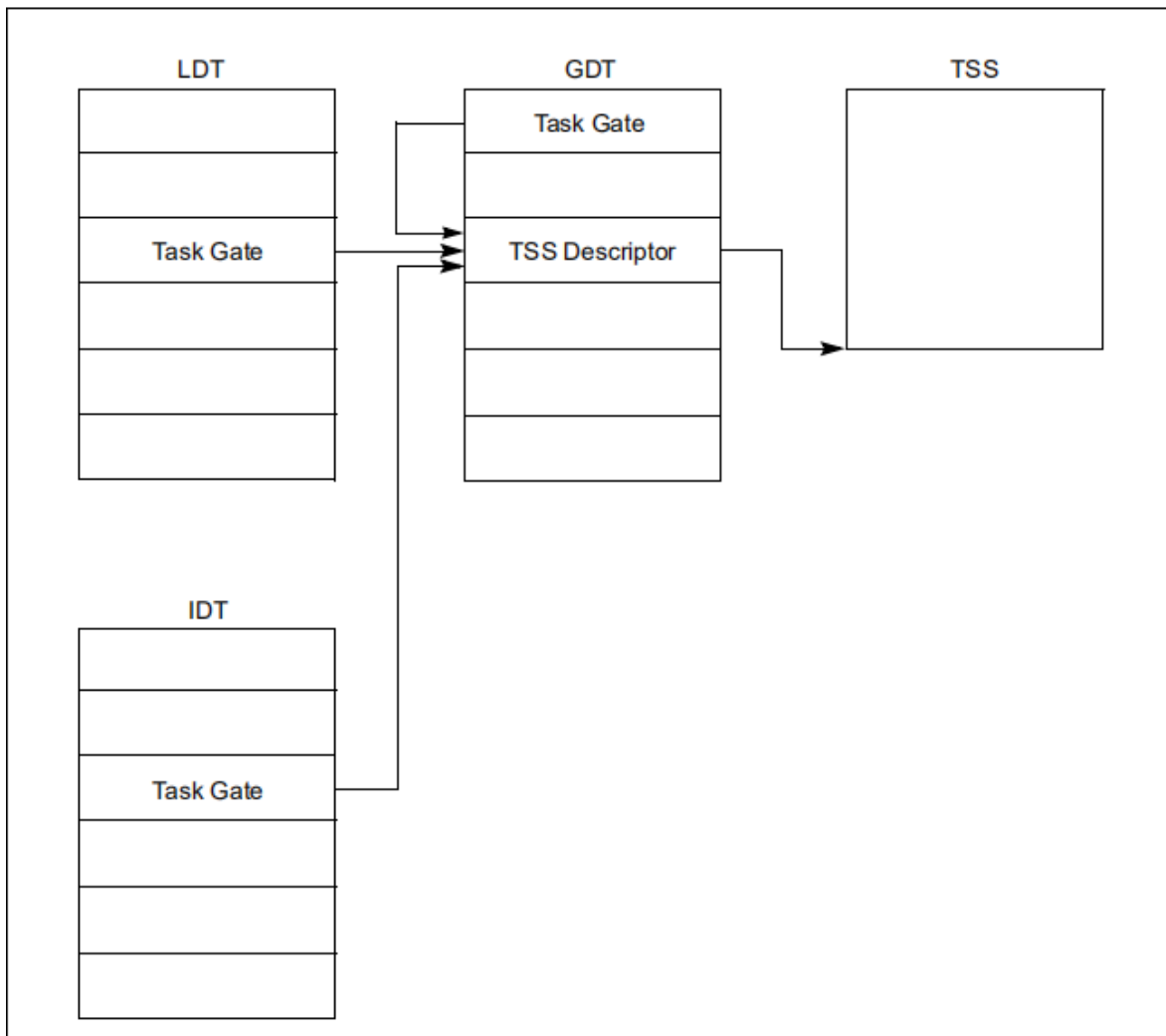


Figure 7-7. Task Gates Referencing the Same Task

描述符

任务门描述符也存储在IDT表中，它的结构如下所示，灰色的部分保留（填0），DPL一般设置为3，方便3环应用程序访问，Type域名是固定的0101，TSS段选择子，指向位于GDT表中TSS段描述符的位置：

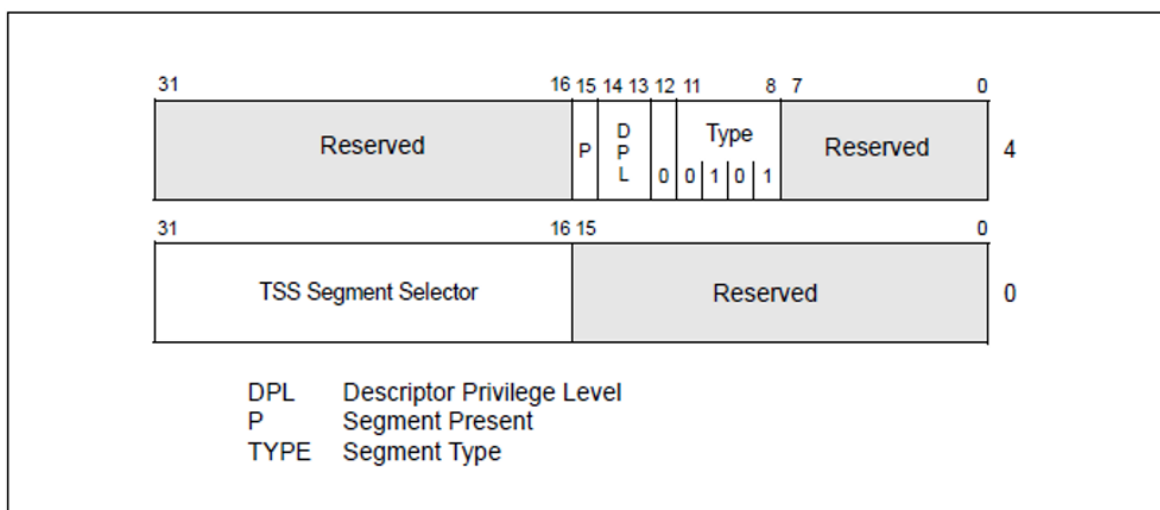


Figure 7-6. Task-Gate Descriptor

实现任务切换

在做"使用任务门实现任务切换"实验之前，我们先来看一下任务门的执行过程如下所示：

1. 使用INT N指令；
2. 查IDT表，找到任务门描述符；
3. 通过任务门描述符的TSS段选择子，在GDT表中找到TSS段描述符；
4. 使用TSS段中的值修改TR寄存器；
5. IRETD返回。

所以按照流程来，我们需要先构建TSS段描述符，这里与任务段实验一样，不过为了论证之前所说的通过任务门找到TSS段描述符就不检查权限这个说法，我们需要将TSS段描述符中的DPL位修改为0，所以得出TSS段描述符为：**00008912`fd780068**，在Windbg中写入GDT表：

```
kd> eq 8003f048 00008912`fd780068
WriteVirtual: 8003f048 not properly sign extended
kd> dq 8003f000
ReadVirtual: 8003f000 not properly sign extended
8003f000  00000000`00000000 00cf9b00`0000ffff
8003f010  00cf9300`0000ffff 00cffb00`0000ffff
8003f020  00cff300`0000ffff 80008b04`200020ab
8003f030  ffc093df`f0000001 0040f300`00000fff
8003f040  0000f200`0400ffff 00008912`fd780068
8003f050  80008955`97000068 80008955`97680068
8003f060  00009302`2f30ffff 0000920b`80003fff
8003f070  ff0092ff`700003ff 80009a40`0000ffff
```

接着构建任务门描述符，根据TSS段描述符的位置得出选择子带入并结合门描述符的结构，最终得出任务门描述符为：**0000e500`004b0000**，在Windbg中写入IDT表：

```

Command
8003f420 804eee00`000801e6 804e8e00`0008034b
8003f430 804e8e00`000804c9 804e8e00`00080b4d
8003f440 00008500`00501188 804e8e00`00080f5a
8003f450 804e8e00`0008107f 804e8e00`000811c4
8003f460 804e8e00`0008142e 804e8e00`0008172d
8003f470 804e8e00`00081e37 804e8e00`00082175
8003f480 804e8e00`0008229b 804e8e00`000823dd
8003f490 804e8500`00a02175 804e8e00`0008254a
8003f4a0 804e8e00`00082175 804e8e00`00082175
8003f4b0 804e8e00`00082175 804e8e00`00082175
8003f4c0 804e8e00`00082175 804e8e00`00082175
8003f4d0 804e8e00`00082175 804e8e00`00082175
8003f4e0 804e8e00`00082175 804e8e00`00082175
8003f4f0 804e8e00`00082175 80708e00`0008410c
8003f500 00000000`00080000 00000000`00080000
8003f510 00000000`00080000 00000000`00080000
kd> eq 8003f500 0000e500`004b0000
WriteVirtual: 8003f500 not properly sign extended
kd> dq 8003f400 L24
ReadVirtual: 8003f400 not properly sign extended
8003f400 804d8e00`0008fabd 804d8e00`0008fc3e
8003f410 00008500`0058112e 804eee00`00080061
8003f420 804eee00`000801e6 804e8e00`0008034b
8003f430 804e8e00`000804c9 804e8e00`00080b4d
8003f440 00008500`00501188 804e8e00`00080f5a
8003f450 804e8e00`0008107f 804e8e00`000811c4
8003f460 804e8e00`0008142e 804e8e00`0008172d
8003f470 804e8e00`00081e37 804e8e00`00082175
8003f480 804e8e00`0008229b 804e8e00`000823dd
8003f490 804e8500`00a02175 804e8e00`0008254a
8003f4a0 804e8e00`00082175 804e8e00`00082175
8003f4b0 804e8e00`00082175 804e8e00`00082175
8003f4c0 804e8e00`00082175 804e8e00`00082175
8003f4d0 804e8e00`00082175 804e8e00`00082175
8003f4e0 804e8e00`00082175 804e8e00`00082175
8003f4f0 804e8e00`00082175 80708e00`0008410c
8003f500 0000e500`004b0000 00000000`00080000
8003f510 00000000`00080000 00000000`00080000

```

接着我们在代码中加入INT N指令，N为0x20（我们写入的任务门描述符在IDT表中过的索引值），完整代码如下：

```

1  #include <stdio.h>
2  #include <windows.h>
3
4  int o_eax, n_esp;
5  short n_cs, n_ss;
6
7  void __declspec(naked) GetValue() {
8      _asm {
9          mov o_eax, eax
10         mov n_esp, esp
11         mov ax, cs
12         mov n_cs, ax
13         mov ax, ss
14         mov n_ss, ax
15         mov eax, o_eax

```



```

16         iret
17     }
18 }
19
20 void main()
21 {
22     char stack[100] = {0}; // 堆栈
23     int iCr3 = 0;
24     printf("Input: ");
25     scanf("%x", &iCr3);
26     getchar();
27
28     DWORD tss[0x68] = {
29         0x00000000, // Previous Task Link
30         // 不同权限对应的ESP、SS，如果不涉及到权限切换所以可以将这些寄存器的值全部填0
31         0x00000000, // ESP0
32         0x00000000, // SS0
33         0x00000000, // ESP1
34         0x00000000, // SS1
35         0x00000000, // ESP2
36         0x00000000, // SS2
37         (DWORD)iCr3, // Cr3, 与页的知识有关，必须要赋值
38         0x00401020, // EIP, 下一次执行代码的位置，必须要赋值
39         0x00000000, // EFLAGS
40         0x00000000, // EAX
41         0x00000000, // ECX
42         0x00000000, // EDX
43         0x00000000, // EBX
44         (DWORD)stack, // ESP, 任务切换时也需要切换堆栈，所以在代码中我们可以声明一个数组，将其地址作为一块堆栈
45         0x00000000, // EBP
46         0x00000000, // ESI
47         0x00000000, // EDI
48         0x00000023, // ES
49         0x00000008, // CS, 切到0环的代码段描述符
50         0x00000010, // SS, CS与SS需要保持一致
51         0x00000023, // DS
52         0x00000030, // FS, 切到0环就是0x30, 3环就是0x3B
53         0x00000000, // GS, Windows没有使用这个段寄存器所以永远是0
54         0x00000000, // LDT, 填0
55         0x20ac0000 // IO_MAP, Windows2000以后不用了
56     };
57
58     _asm {
59         int 0x20
60     }
61
62     // 输出寄存器的值证明完成了任务切换
63     printf("ESP: %x, CS: %x, SS: %x \n", n_esp, n_cs, n_ss);
64
65     getchar();
66 }

```

编译运行程序，在Windbg中输入**!process 0 0**指令找到Cr3值输入到程序中即可，最终我们可以看到获取到的寄存器值确实为我们所构建的TSS中对应的值：

```
53     0x00000000, // GS, Windows没有使用这个段寄存器所以永远是0
54     0x00000000, // LDT, 填0
55     0x20ac0000 // IO_MAP, Windows2000以后不用了
56 };
57
58 _asm {
59     int 0x20
60 }
61
62 // 输出寄存器的值证明完成了任务切换
63 printf("ESP: %x, CS: %x, SS: %x \n", n_esp, n_cs, n_ss);
64
65 getchar();
66 }
67
```

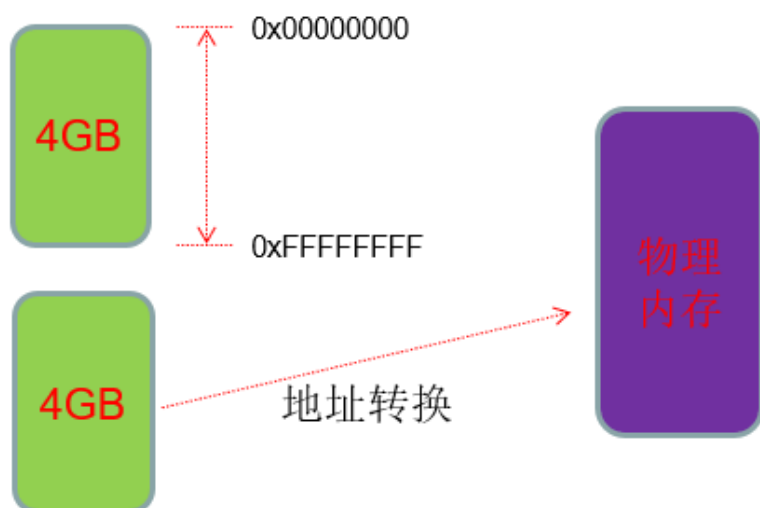


```
C:\Program Files\Microsoft Visual Studio\MyProjects\Test1\Debug\Test1.exe
Input: 69485000
ESP: 12ff1c, CS: 8, SS: 10
```

3 页

保护模式下有段、页机制，本章节就来了解页的机制，它相对于段来说更加重要。

3.1 10-10-12分页

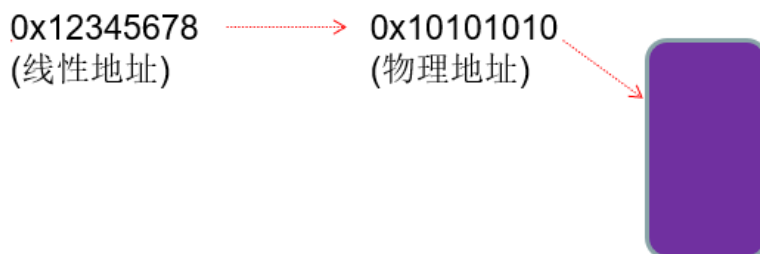


3.1.1 术语概念

为了便于更好的学习，我们需要了解一下线性地址、有效地址、物理地址分别是什么。我们看汇编指令时经常会看见如下类似的指令：

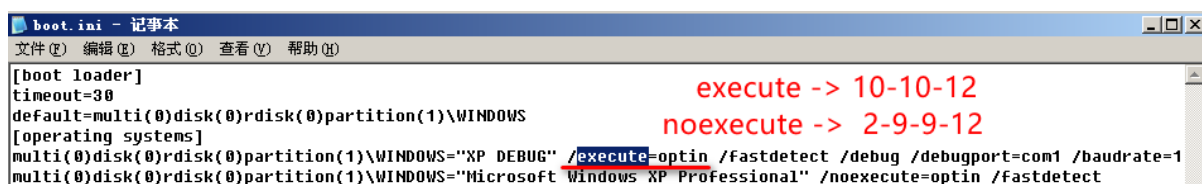
```
1  MOV EAX, DWORD PTR DS:[0x12345678]
```

在这里指令中，**0x12345678就是有效地址**，根据之前段的学习，我们知道这段指令真正读取的地址是 **DS.Base+0x12345678**，**这个地址就是线性地址（有效地址+段寄存器的Base）**。有了线性地址之后CPU会将其转为物理地址，这样才能找到真正的数据。



32位系统上将线性地址转为物理地址有2种方式：10-10-12、2-9-9-12，我们本节要学的就是前者。

我们之前进行双机调试配置时候设置了C:\boot.ini这个文件，在写入的内容中有一个参数，如下图所示，当该参数为execute时则表示当前的分页机制就是10-10-12，noexecute则表示当前的分页机制为2-9-9-12：



```

boot.ini - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

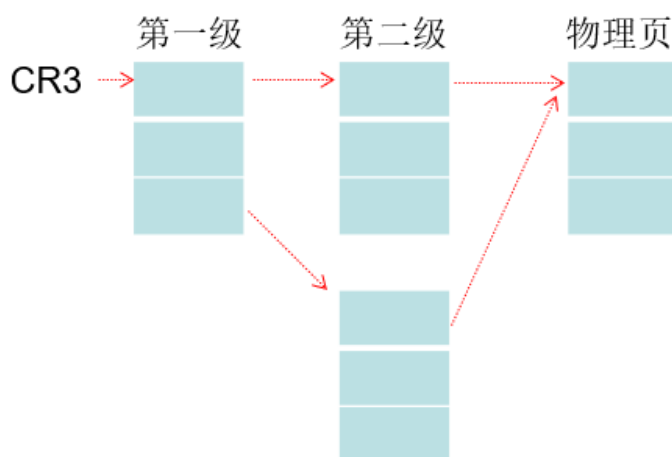
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="XP DEBUG" /execute=optin /fastdetect /debug /debugport=com1 /baudrate=1
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional" /noexecute=optin /fastdetect

```

修改该参数然后重启系统即可。

3.1.2 寻找物理地址

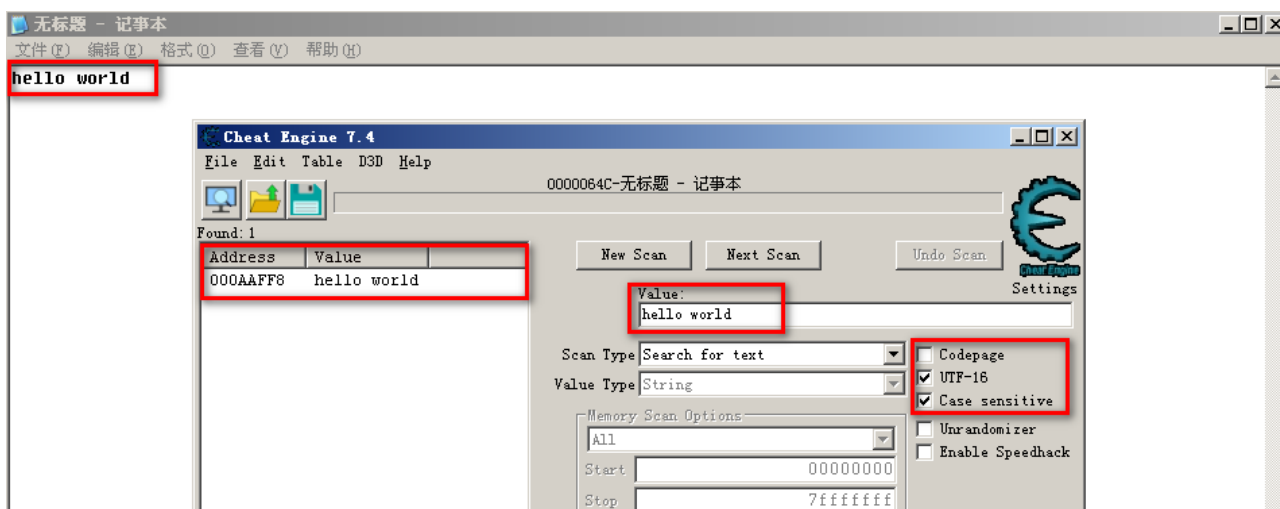
在做实验之前我们需要知道从线性地址寻找到物理地址的流程，如下所示，CPU通过进程的CR3（CR3是一个寄存器，CR3指向的页有4096字节（4KB），每4字节作为一个成员存储，一共有1024个成员）找到第一级页，再通过第一级找到第二级，最后找到物理页：



我们可以将第一级、第二级理解为一个目录，通过目录找到物理页（内容），而目录的索引就是10-10-12分页机制所划分出来的。

所谓10-10-12分页机制，本质上就是将线性地址按位分为三个部分，分别是10位、10位、12位，然后通过这三个部分作为索引找到物理页。

接下来我们可以手动做实验来从线性地址寻找到物理地址，首先打开一个记事本输入一段内容，然后打开CE软件找到该进程，搜索我们输入的内容，找到该字符串对应的线性地址，如下图所示线性地址即为**0x000AAFF8**：



按10-10-12分页机制分为三个部分，前两部分需要*4：

```

1  0000 0000 00    // Hex: 0*4 = 0
2  00 1010 1010    // Hex: AA*4 = 2A8
3  1111 1111 1000 // Hex: FF8

```

接着我们找到该记事本进程的CR3寄存器值：0x16bcf000，使用dd指令（**dd前需要加上感叹号**）加上第一部分的值在第一级页找到第二级页地址：

```
1  !dd 16bcf000+0
```

```

kd> !dd 16bcf000+0
#16bcf000 16d0e867 16b73867 17100867 00000000
#16bcf010 16bc1867 00000000 00000000 00000000
#16bcf020 00000000 00000000 00000000 00000000
#16bcf030 00000000 00000000 00000000 00000000
#16bcf040 00000000 00000000 00000000 00000000
#16bcf050 00000000 00000000 00000000 00000000
#16bcf060 00000000 00000000 00000000 00000000
#16bcf070 00000000 00000000 00000000 00000000

```

获得第二级页地址0x16d0e867，这个地址不是真正的地址，**低12位为属性，需要将其填0**变为0x16d0e000，接着再加上第二部分的值找到物理页地址：

```
1  !dd 16d0e000+2A8
```

```
kd> !dd 16d0e000+2A8
#16d0e2a8 16d45867 16c6a867 16d83867 16c44867
#16d0e2b8 16dc6867 16ec7867 16dc8867 16fc9867
#16d0e2c8 170ca867 16f0b867 16f8c867 2205f867
#16d0e2d8 00000000 00000000 00000000 00000000
#16d0e2e8 00000000 00000000 00000000 00000000
#16d0e2f8 00000000 00000000 00000000 00000000
#16d0e308 00000000 00000000 00000000 00000000
#16d0e318 00000000 00000000 00000000 00000000
```

获取到0x16d45867，低12位置0，地址为0x16d45000，再加上第三部分的值即可获取到物理地址，这里我们可以以字节的形式查看数据：

```
kd> !dd 16d45000+FF8
#16d45ff8 00650068 006c006c 00000000 00000000
#16d46008 00000000 00000000 00000000 00000000
#16d46018 00000000 00000000 00000000 00000000
#16d46028 00000000 00000000 00000000 00000000
#16d46038 00000000 00000000 00000000 00000000
#16d46048 00000000 00000000 00000000 00000000
#16d46058 00000000 00000000 00000000 00000000
#16d46068 00000000 00000000 00000000 00000000
kd> !db 16d45000+FF8
#16d45ff8 68 00 65 00 6c 00 6c 00-00 00 00 00 00 00 00 00 h.e.l.l.....
#16d46008 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
#16d46018 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
#16d46028 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
#16d46038 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
#16d46048 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
#16d46058 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
#16d46068 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
```

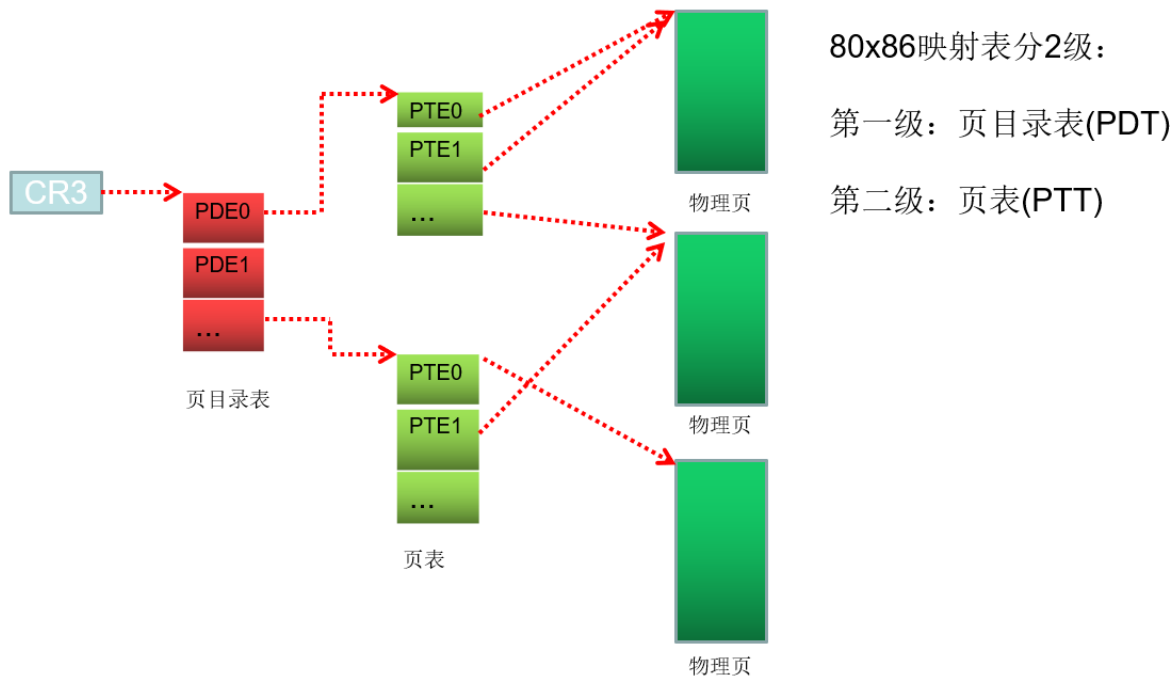
3.1.3 PDE与PTE

术语概念

上一章节中我们简单了解了80x86的10-10-12分页机制，大致的流程我们已经清楚了，但是我们之前所提到的第N级，实际上它们有自己的名字：

第一级叫页目录表（PDT，Page Directory Table），它有4KB的大小，每个成员大小是4字节，每个成员的名字叫页目录项（PDE，Page Directory Entry）；

第二级叫页表（PTT，Page Table），它同样有4KB的大小，每个成员也是4字节大小，每个成员的名字叫页表项（PTE，Page Table Entry）。



10-10-12分页机制可以根据成员数来理解，物理页存储的是数据，以字节为成员单位，物理页有4096个字节，也就是2的12次方，而页表、页目录表成员单位都是4字节，有1024个成员，所以就是2的10次方。

实验

PTE有这几个特征：PTE可以没有物理页；一个PTE只能对应一个物理页；多个PTE可以对应同一个物理页。

为了论证观点，我们可以做一个实验，以线性地址0x0为例，拆分并观察其PTE是否有物理页。

我们可以随便起一个进程然后找到CR3来查看，如下图所示我们看到它没有物理页：

```
Failed to get VadRoot
PROCESS 89379020 SessionId: 0 Cid: 0548 Peb: 7ffd7000 ParentCid: 0234
DirBase: 2c745000 ObjectTable: e1bd5b08 HandleCount: 15.
Image: Test1.exe
```

```
kd> !dd 2c745000+0
#2c745000 2c8b1867 2c6f0867 00000000 00000000
#2c745010 00000000 00000000 00000000 00000000
#2c745020 00000000 00000000 00000000 00000000
#2c745030 00000000 00000000 00000000 00000000
#2c745040 00000000 00000000 00000000 00000000
#2c745050 00000000 00000000 00000000 00000000
#2c745060 00000000 00000000 00000000 00000000
#2c745070 00000000 00000000 00000000 00000000
kd> !dd 2c8b1000+0
#2c8b1000 00000000 00000000 00000000 00000000
#2c8b1010 00000000 00000000 00000000 00000000
#2c8b1020 00000000 00000000 00000000 00000000
#2c8b1030 00000000 00000000 00000000 00000000
#2c8b1040 2c3b2867 00000000 00000000 00000000
#2c8b1050 00000000 00000000 00000000 00000000
#2c8b1060 00000000 00000000 00000000 00000000
#2c8b1070 00000000 00000000 00000000 00000000
```

这也就表示PTE可以没有物理页，那么这个0x0的线性地址我们是否可以读写呢？实际上是可以的，我们已经了解0x0这个地址不能读写的原因是因为其对应PTE没有物理页，那我们可以给它一个物理页。

用如下代码来实验，我们运行这段代码，先获取变量a的线性地址，然后找到其物理页地址，赋值给0x0的PTE，然后回车即可向0x0地址写入内容，并读取：

```

1  #include <stdio.h>
2
3  void main()
4  {
5      int a = 10;
6      printf("a address: %x \n", &a);
7
8      getchar();
9
10     *(int*)0 = 123;
11     printf("0 address data: %x \n", *(int*)0);
12     getchar();
13 }

```

获取变量a的线性地址为0x0012FF7C：

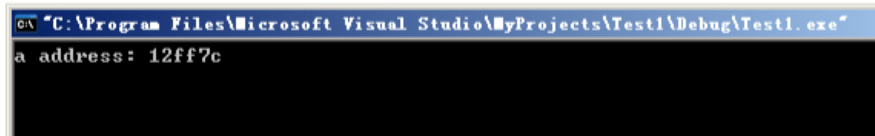
```

void main()
{
    int a = 10;
    printf("a address: %x \n", &a);

    getchar();

    *(int*)0 = 123;
    printf("0 address data: %x \n", *(int*)0);
    getchar();
}

```



按10-10-12分页机制拆分为三部分：

```

1  0000 0000 00 // Hex: 0*4 = 0
2  01 0010 1111 // Hex: 12F*4 = 4BC
3  1111 0111 1100 // Hex: F7C

```

接着我们找到PTE的值为0x25D97867：


```
kd> !dd 25f01000+4BC
#25f014bc 25d97867 14bb7025 14c78025 00000000
#25f014cc 00000000 00000000 00000000 00000000
#25f014dc 00000000 00000000 00000000 00000000
#25f014ec 00000000 00000000 00000000 00000000
#25f014fc 00000000 2615b867 25edc867 262dd867
#25f0150c 00000000 00000000 00000000 00000000
#25f0151c 00000000 00000000 00000000 00000000
#25f0152c 00000000 00000000 00000000 00000000
```

接着我们将该值添加到0x0地址对应的PTE中：

```
1 !ed 25f01000 25d97867
```

```
kd> !ed 25f01000 25d97867
kd> !dd 25f01000+0
#25f01000 25d97867 00000000 00000000 00000000
#25f01010 00000000 00000000 00000000 00000000
#25f01020 00000000 00000000 00000000 00000000
#25f01030 00000000 00000000 00000000 00000000
#25f01040 261c2867 00000000 00000000 00000000
#25f01050 00000000 00000000 00000000 00000000
#25f01060 00000000 00000000 00000000 00000000
#25f01070 00000000 00000000 00000000 00000000
```

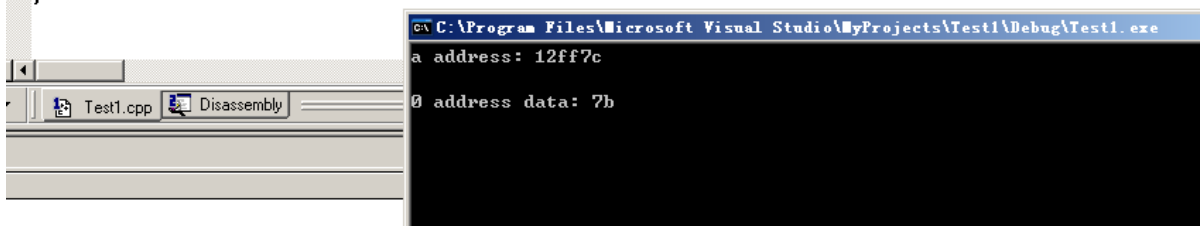
这时候在回到程序按回车执行就会发现我们成功的向0x0线性地址读写了：

```
#include <stdio.h>

void main()
{
    int a = 10;
    printf("a address: %x \n", &a);

    getchar();

    *(int*)0 = 123;
    printf("0 address data: %x \n", *(int*)0);
    getchar();
}
```



低12位属性

在之前的实验中我们知道PDE/PTE的低12位表示属性，这两个属性进行与运算得出的结果就是物理页的属性。

物理页的属性 = PDE属性 & PTE属性



P位

P位表示当前物理页的有效性，为1表示有效，为0则表示无效，**由于物理页的属性是PDE和PTE属性进行与运算的来的所以需要PDE和PTE属性的P位均为1**，才能表示物理页是有效的。

R/W位

R/W位表示物理页的读写权限，为1表示可读可写，为0则表示只读。例如我们之前所学习的常量，它就是一个只读的内容，对应的物理页属性的R/W位值为0，我们可以通过Windbg修改这个值来达到写入的目的。

U/S位

U/S位表示物理页的访问权限，为1表示普通用户可以访问，为0则表示特权用户可以访问。

PS位

PS位（Page Size）只对PDE有意义，为1表示PDE直接指向物理页（没有PTE），低22位就是物理页的页内偏移。因此，线性地址就只能拆分为2部分，**而它的一个物理页就不再是4KB大小，而是4MB大小，我们也称之为大页。**

A位

A位表示当前是否被访问过，为1表示访问过，为0则表示没有访问过。

D位

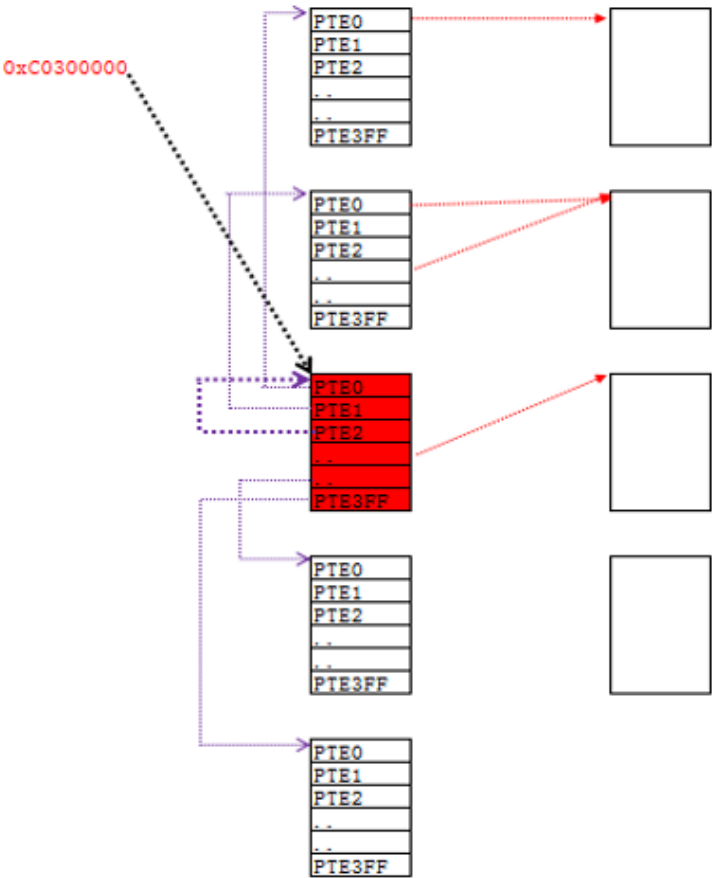
D位表示当前是否被写入过，为1表示写入过，为0则表示没有写入过。

表基址

之前的实验我们可以在Windbg中通过线性地址拆分，找到PDE、PTE，最终找到物理页，对属性进行修改等操作。但是这样的操作都是基于手动调试器的，我们要想通过代码去完成就需要通过表基址（页目录表、页表）。

页目录表

页目录表基址就是0xC0300000，通过它找到的物理页就是页目录表，这个物理页既是页目录表也是页表。这里的项目录表是一张特殊的页表，每一项PTE指向的不是普通的物理页，而是指向其他的页表。



我们可以做个实验来论证这些观点，随便启动一个程序找到它的PDT表：

```
Failed to get VadRoot
PROCESS 89573da0 SessionId: 0 Cid: 0594 Peb: 7ffd9000 ParentCid: 0608
DirBase: 490a4000 ObjectTable: e106d398 HandleCount: 120.
Image: cheatengine-i386.exe
```

```
kd> !dd 490a4000
#490a4000 495f7867 493ed867 49667867 494ee867
#490a4010 00000000 00000000 00000000 4973a867
#490a4020 49811867 49a32867 498b3867 49634867
#490a4030 498b5867 49b76867 49837867 49978867
#490a4040 49ab9867 4997a867 4983b867 4993c867
#490a4050 499fd867 498fe867 495ff867 49a00867
#490a4060 494c1867 498c2867 49883867 49844867
#490a4070 49a45867 49a06867 49a87867 49a08867
```

将0xC0300000按10-10-12分页机制进行拆分：

| | |
|---|---------------------------------|
| 1 | 1100 000 00 // Hex: 300*4 = C00 |
| 2 | 11 0000 000 // Hex: 300*4 = C00 |
| 3 | 0000 0000 0000 // Hex: 0 |

按这个拆分出来的偏移找到物理页，我们就会发现这里找到的物理页内容与上面的PDT表内容是一致的，并且你抛开偏移也会发现，它们的基址都是一样的，所以也就论证了这个物理页既是页目录表也是页表：

```
kd> !dd 490a4000+c00
#490a4c00 490a4863 492a5863 00000000 09e90963
#490a4c10 09e91963 09e92963 09e93963 09e94963
#490a4c20 09e95963 09e96963 09e97963 09e98963
#490a4c30 09e99963 09e9a963 09e9b963 09e9c963
#490a4c40 09e9d963 09e9e963 09e9f963 09ea0963
#490a4c50 09ea1963 09ea2963 09ea3963 09ea4963
#490a4c60 09ea5963 09ea6963 09ea7963 09ea8963
#490a4c70 09ea9963 09eaa963 09eab963 09eac963
kd> !dd 490a4000+c00
#490a4c00 490a4863 492a5863 00000000 09e90963
#490a4c10 09e91963 09e92963 09e93963 09e94963
#490a4c20 09e95963 09e96963 09e97963 09e98963
#490a4c30 09e99963 09e9a963 09e9b963 09e9c963
#490a4c40 09e9d963 09e9e963 09e9f963 09ea0963
#490a4c50 09ea1963 09ea2963 09ea3963 09ea4963
#490a4c60 09ea5963 09ea6963 09ea7963 09ea8963
#490a4c70 09ea9963 09eaa963 09eab963 09eac963
kd> !dd 490a4000+0
#490a4000 495f7867 493ed867 49667867 494ee867
#490a4010 00000000 00000000 00000000 4973a867
#490a4020 49811867 49a32867 498b3867 49634867
#490a4030 498b5867 49b76867 49837867 49978867
#490a4040 49ab9867 4997a867 4983b867 4993c867
#490a4050 499fd867 498fe867 495ff867 49a00867
#490a4060 494c1867 498c2867 49883867 49844867
#490a4070 49a45867 49a06867 49a87867 49a08867
```

页表

基于页目录表基址，我们可以访问某个线性地址的PDT，但是这样仅仅可以操控PDT表的内容（PDE），并不能操控PTE，也就是无法访问PTT表。

我们要想访问PTT表就需要页表基址，页表基址就是0xC0000000，与页目录表基址做的实验一样，我们先找到某个线性地址的PTT表：

Failed to get VadRoot

PROCESS 894208d0 SessionId: 0 Cid: 01f4 Peb: [7ffd8000](#) ParentCid: [00f0](#)
 DirBase: 200a7000 ObjectTable: e1bb6d28 HandleCount: 120.
 Image: cheatengine-i386.exe

```
kd> !dd 200a7000
#200a7000 202ea867 1ff20867 204e9867 1ff61867
#200a7010 00000000 00000000 00000000 200ad867
#200a7020 20153867 20274867 20435867 20236867
#200a7030 20337867 201f8867 20239867 2033a867
#200a7040 2007b867 200fc867 2013d867 200fe867
#200a7050 2023f867 20500867 20281867 203c2867
#200a7060 20283867 1ffc4867 200c5867 20186867
#200a7070 1ff87867 20088867 1ff49867 2008a867
kd> !dd 202ea000
#202ea000 00000000 00000000 00000000 00000000
#202ea010 00000000 00000000 00000000 00000000
#202ea020 00000000 00000000 00000000 00000000
#202ea030 00000000 00000000 00000000 00000000
#202ea040 2026b867 00000000 00000000 00000000
#202ea050 00000000 00000000 00000000 00000000
#202ea060 00000000 00000000 00000000 00000000
#202ea070 00000000 00000000 00000000 00000000
kd> !dd 1ff20000
#1ff20000 1fe62025 1ff9e025 201df025 202c3025
#1ff20010 20004025 20105025 201c6025 1ffc7025
#1ff20020 200c8025 1ff89025 200ca025 20213025
#1ff20030 20194025 20255025 20016025 1ffd6025
#1ff20040 20057025 200d8025 1fe59025 1feda025
#1ff20050 201db025 1fd5c025 1ff5d025 1ff4c025
#1ff20060 2014d025 202ce025 200cf025 201d0025
#1ff20070 00000000 20012025 201d3025 24b61025
```

拆分0xC0000000为三部分，带入查找物理页：

| | |
|---|----------------------------------|
| 1 | 1100 0000 00 // Hex: 300*4 = C00 |
| 2 | 00 0000 0000 // Hex: 0*4 = 0 |
| 3 | 0000 0000 0000 // Hex: 0 |

最终结果如下图所示，我们可以看到查找出来的物理页内容与上图中的PTT表内容是一致的：

```

kd> !dd 200a7000+C00
#200a7c00 200a7863 20268863 00000000 09e90963
#200a7c10 09e91963 09e92963 09e93963 09e94963
#200a7c20 09e95963 09e96963 09e97963 09e98963
#200a7c30 09e99963 09e9a963 09e9b963 09e9c963
#200a7c40 09e9d963 09e9e963 09e9f963 09ea0963
#200a7c50 09ea1963 09ea2963 09ea3963 09ea4963
#200a7c60 09ea5963 09ea6963 09ea7963 09ea8963
#200a7c70 09ea9963 09eaa963 09eab963 09eac963
kd> !dd 200a7000+0
#200a7000 202ea867 1ff20867 204e9867 1ff61867
#200a7010 00000000 00000000 00000000 200ad867
#200a7020 20153867 20274867 20435867 20236867
#200a7030 20337867 201f8867 20239867 2033a867
#200a7040 2007b867 200fc867 2013d867 200fe867
#200a7050 2023f867 20500867 20281867 203c2867
#200a7060 20283867 1ffc4867 200c5867 20186867
#200a7070 1ff87867 20088867 1ff49867 2008a867
kd> !dd 202ea000+0
#202ea000 00000000 00000000 00000000 00000000
#202ea010 00000000 00000000 00000000 00000000
#202ea020 00000000 00000000 00000000 00000000
#202ea030 00000000 00000000 00000000 00000000
#202ea040 2026b867 00000000 00000000 00000000
#202ea050 00000000 00000000 00000000 00000000
#202ea060 00000000 00000000 00000000 00000000
#202ea070 00000000 00000000 00000000 00000000

```

我们也由图得知，**!dd 200a7000+0**指令的结果，0x0开始是第一个PTT表的地址，0x4开始就是第二个PTT表的地址，也就是需要指令变为**!dd 200a7000+4**指令，我们再根据这个指令进行反推：

```

1  !dd 200a7000+C00 // Hex: C00 / 4 = 300, 1100 0000 00
2  !dd 200a7000+4 // Hex: 4 / 4 = 1, 00 0000 0001
3  !dd 1ff20000+0 // Hex: 0, 0000 0000 0000
4
5  -> 1100 0000 0000 0000 0001 0000 0000 0000
6  -> Hex: 0xC0001000

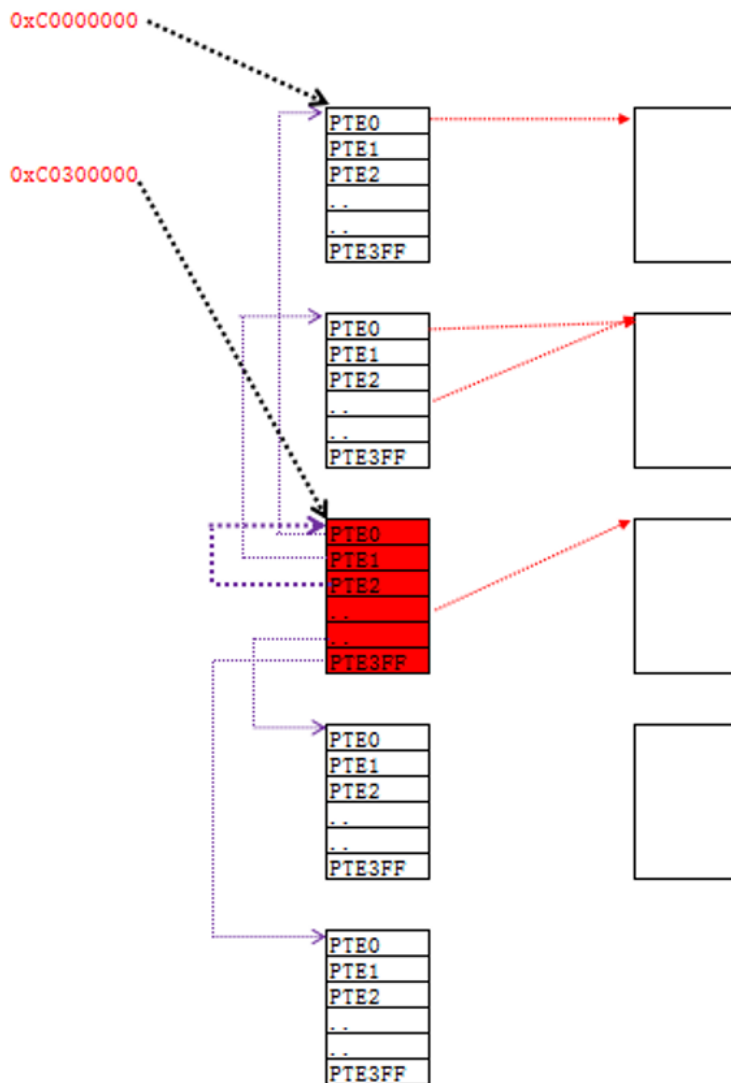
```

最终得出线性地址为：0xC0001000，那么根据这个规律我们就知道在0xC0000000的基础上每递增0x1000，即表示获取下一个PTT，0x1000正好是4096字节（一个页的大小4KB）。

总结

当学习完这部分的内容之后，我们对10-10-12分页机制又有了新的认知：

1. 页表（PTT）被映射到了从0xC0000000到0xC03FFFFFFF的4MB地址空间（一个表4KB，一共有1024个表）；
2. 在这1024个表中有一张特殊的表，就是页目录表（PDT）；
3. 页目录表被映射到了从0xC0300000开始处的4KB地址空间。



掌握了2个表基址，就相当于掌握了一共进程所有的物理内存读写权限，你可以通过如下公式去访问页目录表、页表：

```

1 // 定义
2 10-10-12拆分就是如下：
3 PDI(10)：Page Dictory Index，页目录索引
4 PTI(10)：Page Table Index，页表索引
5 PPI(12)：Physical Page Index，物理页索引
6
7 // 访问页目录表公式
8 0xC0300000 + PDI * 4
9
10 // 访问页表公式
11 0xC0000000 + PDI * 4096 + PTI * 4

```


3.2 2-9-9-12分页

在之前的课程中我们讲解了10-10-12分页机制，在这种机制方式下物理地址最多可达4GB。但随着硬件发展，4GB的物理地址范围已经无法满足需求，Intel在1996年就已经意识到这个问题了，所以设计了新的分页机制，这也就是我们本节课要讲的**2-9-9-12分页机制**，又称为**PAE（物理地址扩展）分页**。

3.2.1 分页设计

在了解2-9-9-12分页机制之前我们需要从10-10-12分页机制开始了解，它们的分页设计的逻辑，这样便于我们掌握和学习。

10-10-12

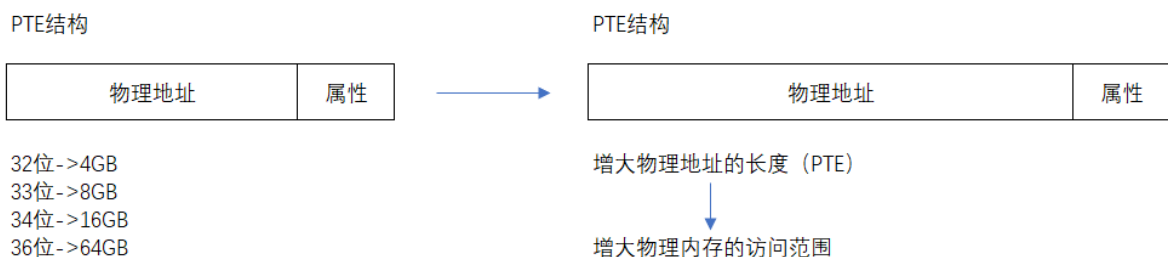
10-10-12的分页设计逻辑如下：

1. Intel认为一张页的大小为4KB是比较合理的，先确定了页的大小为4KB（4096个字节，也就是2的12次方），**此时10-10-12中的12就确定了**；（页内偏移，找到任一个字节）
2. 当初的物理内存比较小，4个字节的PTE成员就够了，页的大小是4KB，所以一个页能存储1024个PTE（也就是2的10次方），**此时10-10-12中的第二个10也确定了**；（寻找PTE）
3. 因为整个线性地址是32位的，我们已经确定了12、10位，那么最后就剩下10位了，**此时10-10-12中的第一个10也就确定了**。（寻找PTI）

2-9-9-12

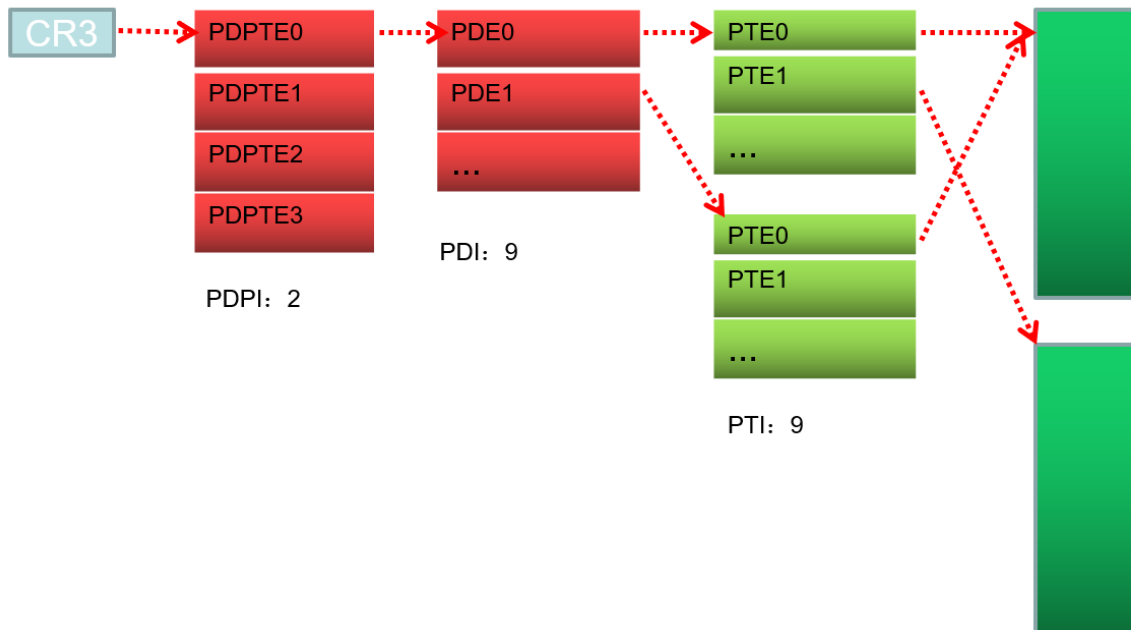
我们再来看2-9-9-12的分页设计逻辑：

1. 页的大小是确定的，4KB大小不能随便更改，**所以2-9-9-12中的12就确定了**；（页内偏移，找到任一个字节）
2. 如果想增大物理内存的访问范围，就需要增大PTE，由于需要考虑对齐，所以增加到8个字节，这里增加的只是PTE结构中的物理地址部分，属性部分并没有增加；那么这里一个页可以存储512个PTE（也就是2的9次方），**此时2-9-9-12中的第二个9确定了**；



3. 同理PDE也由原来的4个字节变成了8个字节，PDT表也由1024个成员变成了512个成员，也就与PTE一样了，**此时2-9-9-12中的第一个9确定了**；
4. 线性地址是32位的，这样算下来就还剩下2位，**这2位就是新拓展出来PDPI（Page Directory Point Table Index，页目录指针表索引）**，也就有了一张新的表叫PDPT（Page Directory Point Table，页目录指针表），其中的成员就是PDPTE（Page Directory Point Table Entry，页目录指针表项，同样是8字节）。

如下图所示就是2-9-9-12的分页结构，在10-10-12分页结构中CR3指向的就是PDT，而在2-9-9-12的分页结构中，在PDT之前，也就是CR3指向的是PDPT表，这张表有四个成员（因为PDPI只有2位，最多能交叉4个结果：00 01 11 10）：



在之前的实验中我们修改了C:\boot.ini文件，现在我们可以修改参数为noexecute，然后重启系统，这样就可以开启2-9-9-12分页机制：

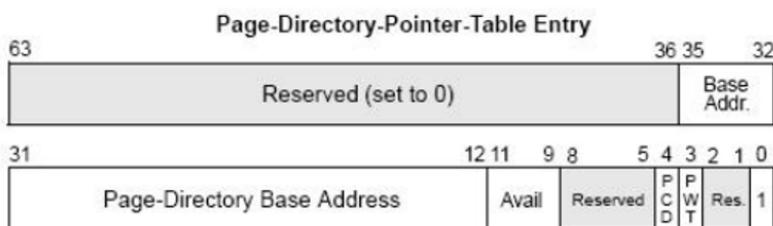
```
boot.ini - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="XP DEBUG" /execute=optin /fastdetect /debug /debugport=com1 /baudrate=1
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional" /noexecute=optin /fastdetect
```

execute -> 10-10-12
noexecute -> 2-9-9-12

3.2.2 PDPTE结构

如下图所示，灰色部分为保留部分，不一定填0。接着我们来看下这个结构，第0位填1；PWT、PCD位需要等到之后的内容学完再学习；第9-11位Available是给操作系统软件用的，CPU不使用；从第12-35位就是页目录表基址。



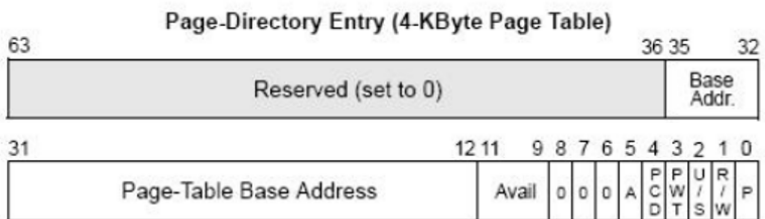
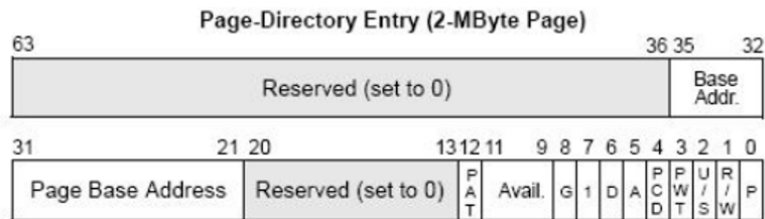
当你想要使用这个地址去寻找PDT表时，与之前10-10-12的实验一样，属性部分（低12位）要填0。

3.2.3 PDE结构

PDE的结构与10-10-12的结构差不多，灰色部分为保留部分，当PS位为1时是大页（下图中的第一个结构），第35-21位是大页的物理地址，这样36位的物理地址的低21位为0，这就意味着页的大小为2MB，且都是2MB对

齐；并且多了一个PAT位（页属性表），也就是第12位，该位与CPU相关，但并不是所有CPU都支持该位，如果不支持，该位会填0。

当P位为0时（下图中的第二个结构），第35-12位是页表基址，低12位填0，共36位。



3.2.4 PTE结构

PTE的结构很简单，第12-35位就是物理页基址，低12位填0，共36位。物理页基址加12位的页内偏移，就可以找到具体的数据。



3.2.5 XD标志位

XD标志位在AMD中称之为NX，即No Exection。它的设计是因为大多数漏洞产生的原因是因为数据被当成指令去执行了，为了防范这一问题，Intel做了硬件保护，做了一个不可执行位。当XD位为1时，你的软件存在溢出漏洞也没关系，因为即使你的EIP跳到了危险的"数据区"，也是不可执行的。

在2-9-9-12分页机制下，PDE与PTE的最高位为XD/NX位：

| | | | | |
|---|----|-------|------|----------|
| X | 保留 | 35-12 | 物理地址 | 低12位（属性） |
|---|----|-------|------|----------|

3.3 TLB

我们都知道通过一个线性地址访问一个物理页，在10-10-12的分页机制下，需要先读PDE在读PTE，最后读4字节的页，这样从本质上来说读取的内容有12个字节，很影响效率；并且在2-9-9-12分页机制下由于物理地址的拓展，会导致读取的字节更多，甚至在某些情况下，数据不在同一物理页，就会存在跨页的情况。

为了提高效率，**CPU在内部做了一张表，来记录这些东西，这就是TLB（Translation Lookaside Buffer，转译查找缓冲区）**，它和寄存器一样快，但同样它的大小也不会太大。

3.3.1 结构

如下图所示就是TLB的结构（TLB中存储的成员结构，不同的CPU下，TLB表的大小是不一样的），线性地址、物理地址我们都知道：

| LA（线性地址） | PA（物理地址） | ATTR（属性） | LRU（统计） |
|------------|----------|----------|---------|
| 0x81010111 | ... | | |

属性（ATTR）是PDPE、PDE、PTE三个属性进行AND运算出来的结果，如果是10-10-12分页机制的话就是PDE、PTE进行AND运算的结果。

Cr3改了就会直接刷新TLB（也就是进程切换，因为在进程切换后TLB原先存储的线性地址和物理地址的对应关系就没有意义了）；由于操作系统中的高2G映射基本不变，所以为了不重复的建立高2G的TLB对应关系，在PDE和PTE中有个标志位G位，它表示其物理页是否为全局页，当该值为1时则不会刷新TLB（只有当PDE的PS位为1时，即其物理页为大页，G位才有效）；当TLB表存满了之后，将根据统计（LRU）信息（统计信息内存储了每个地址的读写情况）将不常用的地址废弃，最近常用的保留下来。

3.3.2 种类

TLB在x86体系的CPU里的实际应用最早是从Intel的486CPU开始的，在x86体系的CPU里，一般都设有如下4组TLB：

1. 缓存一般页表（4KB字节页面）的指令页表缓存（Instruction-TLB）；
2. 缓存一般页表（4KB字节页面）的数据页表缓存（Data-TLB）；
3. 缓存大尺寸页表（2MB/4MB字节页面）的指令页表缓存（Instruction-TLB）；
4. 缓存大尺寸页表（2MB/4MB字节页面）的数据页表缓存（Instruction-TLB）。

3.4 中断与异常

3.4.1 中断

中断通常是由CPU外部的输入或输出设备（硬件）所触发的，供外部设备通知CPU有事情需要处理，因此又称之为中断请求（IRQ-Interrupt Request）。

中断请求的目的是**希望CPU暂时停止执行当前正在执行的程序**，转去执行中断请求所对应的中断处理程序（中断处理程序在哪由IDT表决定）。

80x86有两条中断请求线：

1. 不可屏蔽中断线，称为NMI（NonMaskable Interrupt）；
2. 可屏蔽中断线，称为INTR（Interrupt Require）。

不可屏蔽中断线

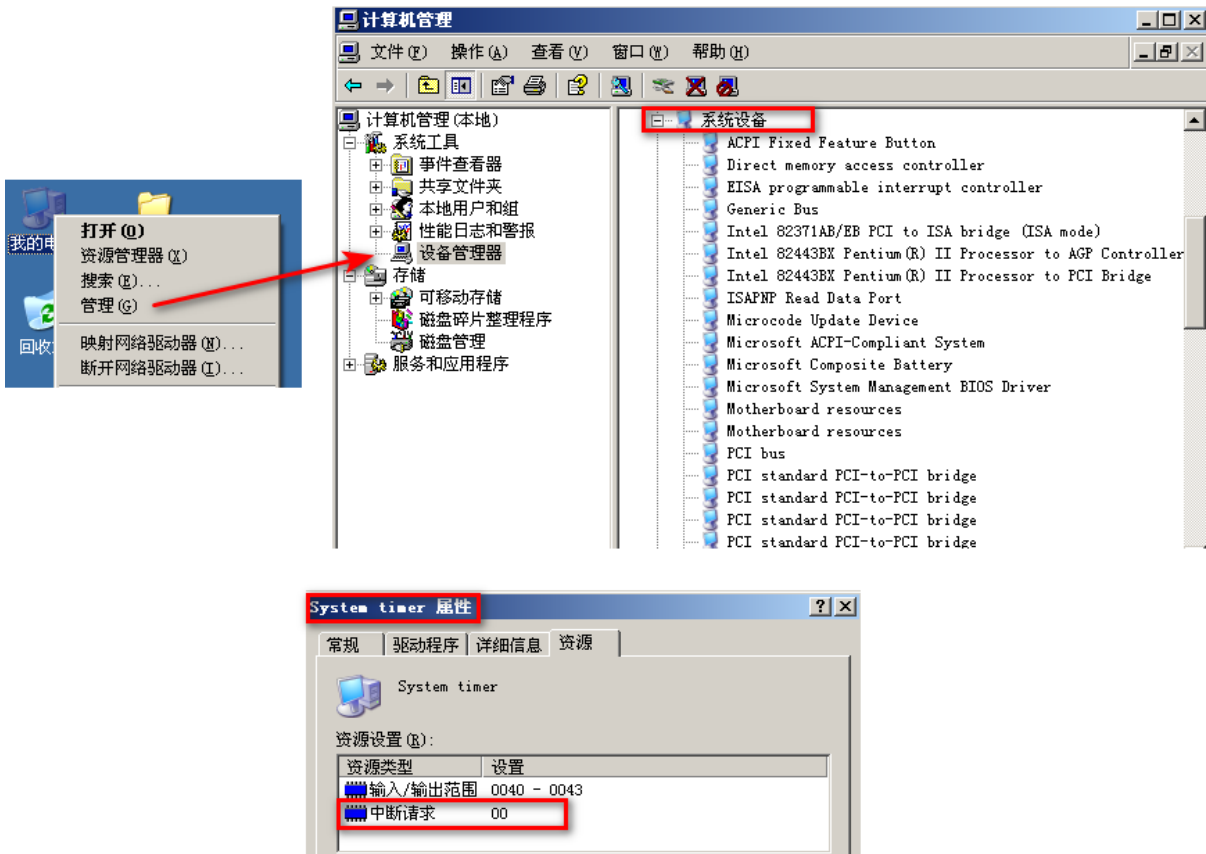
当不可屏蔽中断产生时，CPU会在IDT表中找到下标为0x2的门，通过这个门就可以找到中断处理程序；不可屏蔽中断不受EFLAG寄存器中的IF位的影响，当产生时，CPU必须处理。

| (IDT表)中断号 | NMI | 说明 |
|-----------|--------|--------------|
| 0x2 | 不可屏蔽中断 | 80x86中固定为0x2 |

可屏蔽中断线

在硬件级层面，可屏蔽中断是由一块专门的芯片来管理的，我们称之为中断控制器；它负责分配终端资源和管理各个中断源发出的中断请求；为了便于标识各个中断请求，中断管理器通常在IRQ（Interrupt Request）后面加上数字来表示不同的请求，例如在Windows中时钟中断的IRQ编号为0，也就是IRQ0。

我们可以在通过这个路径：右键我的电脑-管理-设备管理器-System timer-属性-资源，找到Windows时钟的中断编号（其他设备也是一样的）：



当可屏蔽中断产生时，CPU会在IDT表中找到下标为0x30/0x31-0x3F的门，通过这个门就可以找到中断处理程序，如下图所示，时钟中断和其硬件设备的中断下标是不一样的：

| (IDT表) 中断号 | IRQ | 说明 |
|------------|------------|-----------|
| 0x30 | IRQ0 | 时钟中断 |
| 0x31-0x3F | IRQ1-IRQ15 | 其他硬件设备的中断 |

如果我们自己的程序执行时不希望CPU去处理这些中断，可以使用CLI指令清空EFLAG寄存器中的IF位，或使用STI指令设置EFLAG寄存器中的IF位。

需要注意，**硬件设备的中断与IDT表中的对应关系不是固定的**，可以参考APIC（Advanced Programmable Interrupt Controller，高级可编程中断控制器）。

3.4.2 异常

异常通常是CPU在执行指令时检测到的某些错误，比如除0、访问无效页面等。

中断与异常的区别：

1. 中断来自于外部设备，是中断源（键盘、鼠标等等）发起的，CPU是被动触发的；
2. 异常来自于CPU本身，是CPU主动产生的；

之前我们所了解的**INT N指令虽然被称之为软件中断，但其本质是异常**，EFLAG的IF位对该指令是无效的。

异常处理

无论是由硬件设备触发的中断请求还是由CPU产生的异常，处理程序都在IDT表中。

常见的异常处理程序如下：

| 错误类型 | (IDT表) 中断号 |
|------|------------|
| 页错误 | 0xE |
| 段错误 | 0xD |
| 除零错误 | 0x0 |
| 双重错误 | 0x8 |

缺页异常

缺页异常在操作系统中是无时不刻都在发生的，比如当**PDE/PTE的P位为0**或**PDE/PTE的属性为只读但程序试图写入**，这时候就会产生缺页异常。

一旦发生缺页异常，CPU就会执行IDT表中0xE号中断处理程序，也就是由操作系统来接管。



PSE = 1

| | | | | | |
|----------|------|-----|----------|------|-----|
| 10-10-12 | PS=1 | 4M页 | 2-9-9-12 | PS=1 | 2M页 |
| | PS=0 | 4K页 | | PS=0 | 4K页 |

PSE = 0

| | | | | | |
|----------|------|-----|----------|------|-----|
| 10-10-12 | PS=1 | 4K页 | 2-9-9-12 | PS=1 | 4K页 |
| | PS=0 | 4K页 | | PS=0 | 4K页 |

3.6 PWT与PCD位

在之前学习PDE、PDT结构中我们没有讲解PWT、PCD位，现在我们来了解以下。

3.6.1 CPU缓存

在了解这两个位之前我们需要有一个前置知识，那就是CPU缓存。CPU缓存是位于CPU和物理内存之间的临时存储器，它的容量比内存小很多，但是交换速度比内存快很多。

CPU缓存可以做的很大，我们听着它似乎与TLB有着某些相似之处，但实际上是由差异的：TLB缓存的是线性地址和物理地址的对应关系，CPU缓存的是物理地址和实际内容的对应关系。

TLB:

线性地址 <-----> 物理地址

CPU缓存:

物理地址 <-----> 内容

3.6.2 PWT/PCD

PWT（Page Write Through，页直写），该值为1时不仅写入缓存中，也会写入到内存中；为0时只会写入缓存。

PCD（Page Cache Disable，页缓存可编辑），该值为1时禁止写入缓存，直接写入内存；比如页表所在的页已经存储在TLB中了，并不需要再缓存了，所以该位就可以设为1。