1. 段的机制

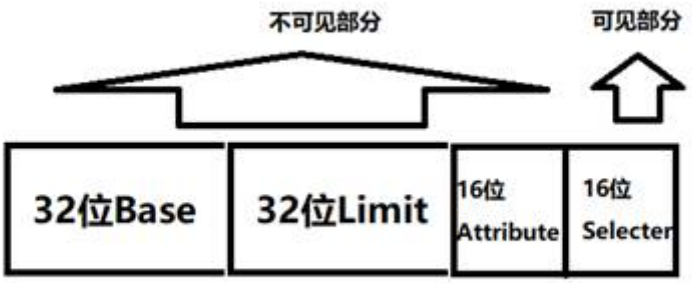
（1）什么是段寄存器

Mov dword ptr ds:[0x123456],eax

我们真正读写的地址 ds.base + 0x123456

段寄存器 ES CS SS DS FS GS LDTR TR 共8个

（2）共计96位，可以通过MOV指令进行读写（LDTR和TR除外）



只读16位，写入96位

（3）段寄存器的属性

BASE属性

_asm{

mov ax , FS

mov CS , ax

mov eax , CS: [0]

}

默认段寄存器0地址不可读取写入，但将FS段寄存器

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	0x7FFDE000	0xFFF
GS	-	-	-	-

_asm{

mov ax , FS

mov gs , ax

mov eax , CS: [0x1000]

}

报错：0xFFF越界

_asm

{

Mov ax , fs

Mov gs , ax

Mov eax , dword ptr ds:[0x7FFDF000 + 0x1000]

Mov dword ptr ds:[var] , eax

}

不会报错，由于ds的limit为0xFFFFFFFF。

（4）段描述符与段选择子

段描述符

1.GDT（全局描述符表）LDT（局部描述符表）

Mov DS , ax

cpu会查表，根据AX的值来决定查找GDT还是LDT

2.查询GDT地址

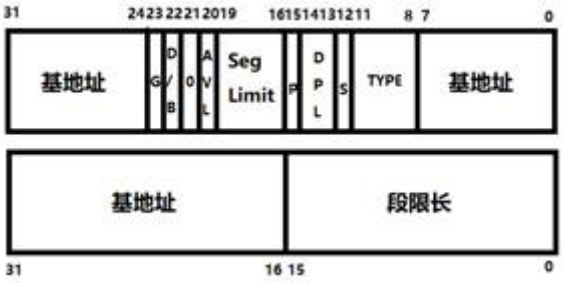
指令 r gdtr

```
kd> r gdtr
gdtr=80b95000
```

显示内容 dq 80b95000

```
kd> dq 80b95000
80b95000  00000000`00000000 00cf9b00`0000ffff
80b95010  00cf9300`0000ffff 00cffb00`0000ffff
80b95020  00cff300`0000ffff 80008b1e`500020ab
80b95030  84409317`ec003748 0040f300`00000fff
80b95040  0000f200`0400ffff 00000000`00000000
80b95050  84008917`c0000068 84008917`c0680068
80b95060  00000000`00000000 00000000`00000000
80b95070  800092b9`500003ff 00000000`00000000
```

GDT表中每一个成员都是8字节的段描述符



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

RPL:Requested Privilege Level (RPL) 请求特权级别

TI:　TI = 0 ---> 查询GDT

TI = 1 ---> 查询LDT

Index:GDT表中的索引

（5）加载段描述符至段寄存器

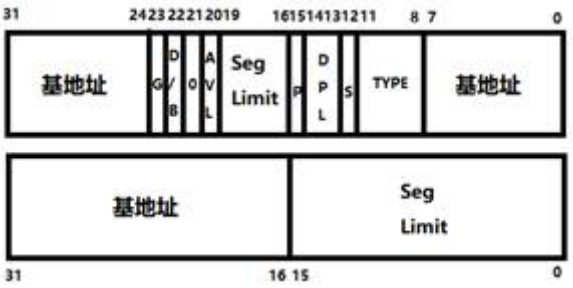
除了mov指令，我们还可以使用LES、LSS、LDS、LFS、LGS指令修改寄存器

CS不能通过上述的指令进行修改，CS位代码段，CS的改变会导致EIP的改变，要改CS,必须要保证CS与EIP一起改

```
Char buffer[6];
_asm
{
Les ecx,fword ptr ds:[buffer] //高2个字节给es，低四个字节给ecx
}
```

注意：RPL<=DPL(在数值上)

（6）段描述符属性



P位:P = 1段描述符有效

P = 0段描述符无效

CPU会优先检查P位，如果无效，不会进行后续检查

G位：

一个段描述符只有64位，如何从64位变成80位

Limit由两部分组成，共计20位，0xFFFFF，还缺失12位

此时，需要了解G位

如果G位为0，此时Limit单位为字节 0x000FFFFF

如果G位为1，此时Limit单位为4kb

4kb = 4 * 1024 = 4096 - 1 = 4095（从0起始）= 0xFFF

此时算上20位的限制 即为0xFFFFF FFF 有0xFFFFF（20位）个0xFFF（4kb）

（寻址到每一个以4字节为单位的空间，前5个F相当于数组序号，后三个F 平摊到每个字节）

1B所对应的段描述符结构体

0001 1011

WORD Selector = 1B;

WORD Atrribute = cf93;

DWORD Base = 00000000;

DWORD Limit = 0xFFFFFFFF;

23所对应的段描述符结构体

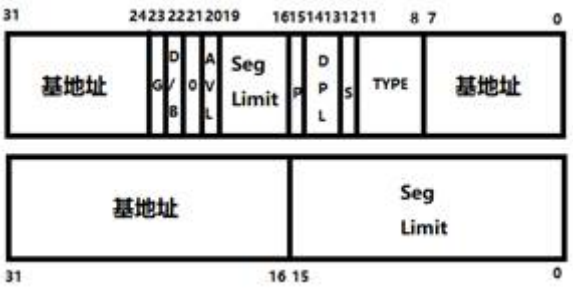
0010 0011

WORD Selector = 23;

WORD Atrribute = cffb;

DWORD Base = 00000000;

DWORD Limit = 0xFFFFFFFF;



S位：

S = 1 代码段或者数据段描述符

S = 0 系统段描述符

TYPE域：

当S = 1时

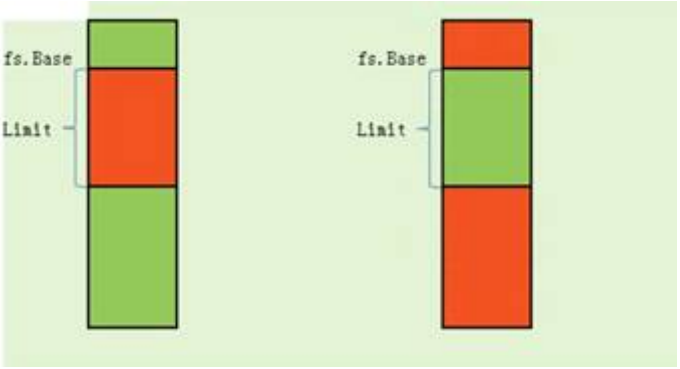
Table 3-1. Code- and Data-Segment Types

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

当第11位等于0时，表示为数据段

E位：拓展位，向上拓展即为0，向下拓展即为0

左图向上拓展　　右图向下拓展



W位：是否可写，可写即为1，不可写即为0.

A位：表示是否被访问过，被访问过即为1，没被访问过即为0.

当第11位等于1时，表示为代码段.

C位：一致位

当C = 1，一致代码段

当C = 0，非一致代码段

权限检查

如果是非一致代码段，要求：CPL == DPL 并且RPL <=DPL

如果是一致代码段，要求：CPL >= DPL

详见代码段跨段执行

R位：表示是否可读，可读即为1，不可读即为0.

A位：表示是否被访问过，被访问过即为1，没被访问过即为0.

当S = 0时

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

Type Field					
Decimal	11	10	9	8	Description
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：

情况一：对CS段的影响

D = 1采用32位寻址方式

D = 0 采用16位寻址方式

Opcode前缀 67 改变寻址方式

情况二：对SS段的影响

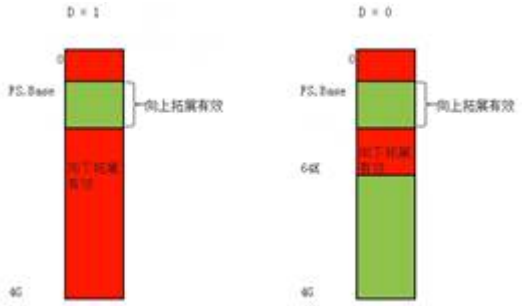
D = 1隐式堆栈访问指令（PUSH POP CALL），使用32位堆栈指针寄存器ESP

D = 0 隐式堆栈访问指令，（PUSH POP CALL）使用16位堆栈指针寄存器SP

情况三：向下扩展的数据段

D = 1 段上线为4GB

D = 0 段上线为64KB



2.段权限检查（数据段）：

1.cpu分级：应用层程序跑在3环，系统程序跑在0环

如何检查当前运行环境在几环？

查看CS段

（例）

当前CS段寄存器 段选择子为 23 即 0010 0011

CS段后两位 11 被称为CPL，当前运行权限级别（CS与SS的后两位永远相同）

2.DPL（Descriptor Privilege Level）描述符特权级别

DPL存储在段描述符中，规定了访问该段所需要的特权级别是什么

通俗的理解：

如果你想访问我，那么你应该具备什么特权

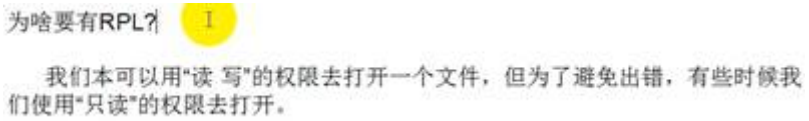
（例） mov DS , AX

如果AX指向的段DPL = 0 但当前程序的CPL = 3 这行指令是不会成功的

3.RPL（Request Privilege Level）请求特权级别

段选择子后两位

RPL的存在是为了防止错误操作



4.数据段的权限检查

（例）CPL = 0 //当前程序处于0环

Mov ax,000B //1011 RPL = 3

Mov ds , ax //ax指向的段描述符的DPL = 0

CPL <= DPL 并且 RPL <= DPL（数值上的比较），系统会根据段描述符判断是数据段还是代码段

注：代码段及系统描述符权限检查 与 数据段 并不相同

3. 代码跨段执行（代码段）

（段描述符仅仅是描述段空间属性特征的八字节数据，更多的是设置权限级别，对同一段，可以附加不同的段描述符从而修改其属性）

1.代码间的跳转（段间跳转）

JMP 0x20:0x004183D7

CPU如何执行这行代码？

（1）段选择子拆分

0x20 对应二进制形式 0010 0000

RPL = 00

TI = 0

Index = 4

（2）查表得到段描述符

TI = 0 查询GDT表

Index = 4 找到对应的段描述符（注：如果该段为数据段，无法跳转）

四种情况可以跳转：代码段、调用门、TSS任务段、任务门

（3）权限检查

如果是非一致代码段，要求：CPL == DPL 并且RPL <=DPL

如果是一致代码段，要求：CPL >= DPL （高权限禁止访问）

换句话说，对于数据段来说，只要RPL CPL <= DPL 便可以访问数据

对于代码段来说，一致代码段 CPL >= DPL 低权限皆可访问

非一致代码段 CPL == DPL 并且 RPL <= DPL

（4）加载段描述符

通过上面的的权限检查后，CPU会将描述符加载到CS段寄存器中

（5）代码执行

CPU将CS.Base + Offset的值写入EIP,然后执行CS：EIP处的代码，段间跳转结束

4. 长调用与短调用

1. 短调用

指令格式：CALL 立即数/寄存器/内存

修改ESP及EIP

2. 长调用（跨段不提权）

指令格式：CALL CS:EIP(EIP是废弃的)

同权限间的跳转

发生改变的寄存器：ESP EIP CS

3. 长调用（跨段并提权）

指令格式：CALL CS:EIP(EIP是废弃的)

提权跳转

发生改变的寄存器：ESP EIP CS SS

跨段并提权会导致堆栈也发生切换，所以0环的堆栈环境会依次将3环的SS、ESP、CS、返回地址压入栈中。

4. 总结

(1)跨段调用时，一旦有权限转换，就会切换堆栈

(2)CS的权限一旦改变，SS的权限也要随着改变，CS与SS的权限等级必须一样

(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权限

5. 调用门

1. 调用门的执行流程

指令格式：CALL CS:EIP(EIP是废弃的)

执行步骤

（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

（2）在调用门描述符中存储另一个代码段段的选择子

（3）选择子指向的段段.Base + 偏移地址 就是真正要执行的地址

2.门描述符

(1)S位为0

(2)

真正要执行的地址=提权的段选择子指向的段描述符的段基地址+新地址的偏移

```
#include "stdafx.h"
#include <windows.h>

void __declspec(naked) GetRegister()
{
    _asm
    {
        int 3

        retf                //注意返回,不能是ret
    }
}

int main()
{

    char buff[6];

    *(DWORD*)&buff[0] = 0x12345678;
    *(WORD*)&buff[4] = 0x48;
    _asm
    {
        call fword ptr[buff]
    }
    getchar();
    return 0;
}
```

构造的调用门0000EF00 00080000

红色数字填写执行代码的虚拟地址

通过调用门进行长调用（提权）后，被修改的寄存器SS , ESP , CS, 返回地址（按照压栈的顺序）

3.调用门总结

(1)当通过门，权限不变的时候，只会PUSH两个值：CS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的值由调用门决定 新的SS和ESP由TSS提供

(3)通过门调用时，要执行哪行代码由调用门决定，但使用RETF返回时，由堆栈中压入的值决定，这就是说，进门时只能按指定路线走，出门时可以翻墙（只要改变堆栈里面的值就可以想去哪就去哪）

(4)可不可以再建个门出去呢？也就是用CALL 当然可以了 前门进 后门出

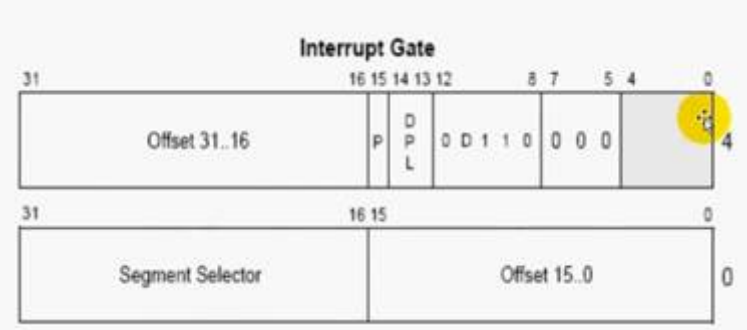
6. 中断门

1. IDT

(1)　IDT即中断描述符表，同GDT一样，IDT也是由一系列描述符组成，每个描述符占8个字节，IDT表中第一个元素不是NULL

```
kd> r idtr
idtr=80b95400
kd> dq 80b95400
```

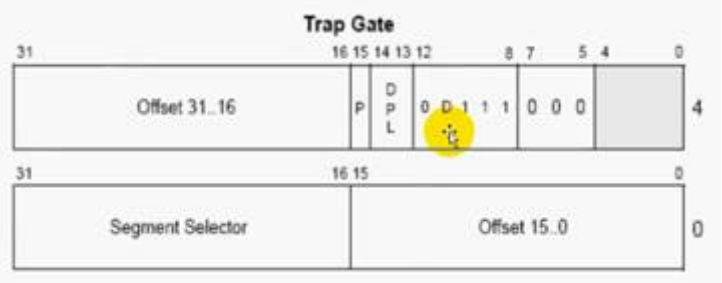
(2)



基本与调用门一样，只不过不再允许传递参数

7. 陷阱门

1. 查询IDT表



除了TYPE ,其余与中断门基本一致

2. 陷阱门与中断门的区别

(1)　中断门执行时，将IF位清零，但陷阱门不会

IF = 1　　会接收到可屏蔽中断

IF = 0　　不会接收到可屏蔽中断

8. 任务段

1. TSS是一块内存，共计104字节

2. 当CS段与SS段发生权限切换时，会从TSS任务段中取出ESP以及SS段选择子（0环的ESP及SS的段选择子）

3. 如何获取TSS地址？TR寄存器中存储着TSS段的地址以及Limit

在系统启动时，TR寄存器中的值是由GDT表中加载进去的

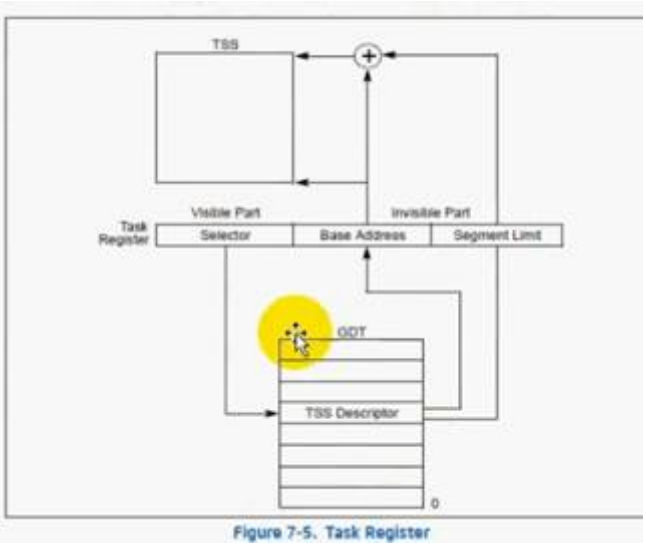


Figure 7-5. Task Register

指令：LTR 将TSS段描述符加载到TR寄存器

指令：STR 读TR寄存器（只读TR的16位段选择子）

9. 任务门

1.任务门的执行流程

(1) INT N

(2) 查IDT表，找到中断门描述符

(3) 通过中断门描述符，查GDT表，找到任务段描述符

(4) 使用TSS段中的值修改寄存器

(5) IRETD返回