保护模式 对 CPL，RPL，DPL 的总结

学习过程中遇到一个对保护模式总结很好的 Blog，转来分享一下。

先说下特权级的概念,在保护模式下,系统依靠特权级来实施代码和数据的保护,相当于权限啦。特权级共有 4 个级别,0,1,2,3,数字越小表示权限越高。如图:

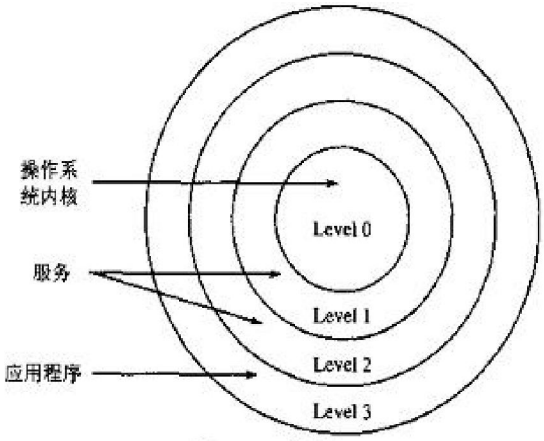

图 3-10 特权级

较为核心的代码和数据放在较高(靠内)的层级中,处理器用此来防止较低特权的任务在不被允许的情况下访问处于高特权级的段。为了防止概念混淆，我们不用特权级大小来说明，改为内层(高)，外层(低)来讲。

特权级有 3 种：CPL,DPL 和 RPL，每个都是有 4 个等级。

我对他们的 理解是这样：一般来说，CPL 代表当前代码段的权限，如果它想要去访问一个段或门，首先要看看对方的权限如何，也就是检查对方的 DPL，如果满足当前的权限比要访问的权限高，则有可能允许去访问，有些情况我们还要检查选择子的权限，即 RPL，因为我们通过选择子:偏移量的方式去访问一个段，这算是一个访问请求动作，因此称为请求访问权限 RPL(Requst Privilege Level)。当请求权限也满足条件，那么访问就被允许了。

CPL(Current Privilege Level)

CPL 是当前执行的任务的特权等级,它存储在 CS 和 SS 的第 0 位和第 1 位上。(两位表示 0～3 四个等级)
通常情况下，CPL 等于代码所在段的特权等级，当程序转移到不同的代码段时，处理器将改变 CPL。
注意:在遇到一致代码段时，情况特殊，一致代码段的特点是：可以被等级相同或者更低特权级的代码访问，当处理器访问一个与当前代码段 CPL 特权级不同的一致代码段时，CPL 不会改变。

DPL(Descriptor Privilege Level)

表示门或者段的特权级，存储在门或者段的描述符的 DPL 字段中。正如上面说的那样，当当前

代码段试图访问一个段或者门时，其 DPL 将会和当前特权级 CPL 以及段或门的选择子比较，根据段或者门的类型不同，DPL 的含义不同：

1.数据段的 DPL：规定了访问此段的最低权限。比如一个数据段的 DPL 是 1，那么只有运行在 CPL 为 0 或 1 的程序才可能访问它。为什么说可能呢？因为还有一个比较的因素是 RPL。访问数据段要满足有效特权级别（上述）高于数据段的 DPL.

2.非一致代码段的 DPL(不使用调用门的情况)：DPL 规定访问此段的特权，只有 CPL 与之相等才有可能访问。

3.调用门的 DPL，规定了程序或任务访问该门的最低权限。与数据段同。

4.一致代码段和通过调用门访问的非一致代码段，DPL 规定访问此段的最高权限。

比如一个段的 DPL 为 2，那么 CPL 为 0 或者 1 的程序都无法访问。

5. TSS 的 DPL，同数据段。

RPL（Rquest Privilege Level）

RPL 是通过选择子的低两位来表现出来的(这么说来，CS 和 SS 也是存放选择子的，同时 CPL 存放在 CS 和 SS 的低两位上，那么对 CS 和 SS 来说，选择子的 RPL＝当前段的 CPL)。处理器通过检查 RPL 和 CPL 来确认一个访问是否合法。即提出访问的段除了有足够的特权级 CPL，如果 RPL 不够也是不行的(有些情况会忽略 RPL 检查)。

为什么要有 RPL？

操作系统往往通过设置 RPL 的方法来避免低特权级的应用程序访问高特权级的内层数据。

例子情景：调用者调用操作系统的某过程去访问一个段。

当操作系统(被调用过程)从应用程序(调用者)接受一个选择子时，会把选择子的 RPL 设置称调用者的权限等级，于是操作系统用这个选择子去访问相应的段时(这时 CPL 为操作系统的等级，因为正在运行操作系统的代码)，处理器会使用调用者的特权级去进行特权级检查，而不是正在实施访问动作的操作系统的特权级(CPL)，这样操作系统就不用以自己的身份去访问(就防止了应用去访问需要高权限的内层数据,除非应用程序本身的权限就足够高)。

那么 RPL 的作用就比较明显了：因为同一时刻只能有一个 CPL，而当低权限的应用去调用拥有至高权限的操作系统的功能来访问一个目标段时，进入操作系统代码段时 CPL 变成了操作系统的 CPL，如果没有 RPL，那么权限检查的时候就会用 CPL，而这个 CPL 权限比应用程序高，也就可能去访问需要高权限才能访问的数据，这就不安全了。所以引入 RPL，让它去代表访问权限，因此在检查 CPL 的同时，也会检查 RPL.一般来说如果 RPL 的数字比 CPL 大(权限比 CPL 的低)，那么 RPL 会起决定性作用。

说这么多不明白都不行啦~真累

下面是引用的一个超棒的关于权限控制的总结：

引用地址

还有一篇文章在此。

■ 数据访问时的权限 check

一、 访问 data segment 时（ds、es、fs 及 gs）

1、 程序指令要访问数据时，data segment selector 被加载进 data segment register（ds、es、fs 和 gs）前，处理器会进行一系列的权限检查，通过了才能被加载进入 segment register。处理器分为两步进行检查：

★ CPL（当前程序运行的权限级别）与 RPL（位于 selector 中的 RPL）作比较，并设置有效权限级别为低权限的一个。

★ 得出的有效权限级别与 DPL（segment descriptor 中的 DPL）作比较，有效权限级别高于 DPL，那么就通过。低于就不允许访问。

2、举个例子：

如果：CPL = 3、RPL = 2、DPL = 2。那么

```
EPL = CPL > RPL ? CPL : RPL;
if (EPL <= DPL) {
/* 允许访问 */
} else {
/* 失败， #GP 异常生产，执行异常 */
}
或者：
if ((CPL <= DPL) && (RPL <= DPL)) {
/* 允许访问 */
} else {
/* 失败， #GP 异常生产，执行异常 */
}
```

也就是要访问目标 data segment，那么必须要有足够的权限，这个足够的权限就是：当前运行的权限级别及选择子的请求权限级别要高于等于目标 data segment 的权限级别。

二、 访问 stack segment 时

1、 该问 stack 时的权限检查更严格，CPL、RPL 及 DPL 三者必须相等才能通过该问请求。

2、 举个例子：

```
if (CPL == RPL && RPL == DPL && CPL == DPL) {
      /* 允许访问 */
} else {
     /* 失败， #GP 异常生产，执行异常 */
}
```

也就是说每个权限级别有相对应的 statck segment。不能越权访问，即使高权限访问低权限也是被拒绝的

■ 控制权的转移及权限检查。

权限检查的 4 个要素:

★ CPL：处理器当前运行的级别，也就是：当前 CS 的级别，在 CS 的 BIT0 ~ Bit1

★ **DPL**：访问目标代码段所需的级别。定义在 segment descriptor 的 **DPL** 域中

★ **RPL**: 通过 selector 进行访问时，selector 内定义的级别。

★ conforming/nonconforming：目标代码属于 nonconforming 还是 conforming 定义在 segment descritptor 的 C 标志位中

x86 的各方面检查依赖于目标代码段是 nonconforming(不一致) 还是 conforming(一致) 类型

一、 直接转移（far call 及 far jmp）

1、 直接转移定义为不带 gate selector 或 taskselector 的远调用。当执行一条 call cs:eip 或 jmp cs:eip 指令时，cs 是目标代码段的 selector，处理器在加载指令操作数中的 cs 进 cs register 前，要进行一系列的权限检查，控制权的转移权限分两部分，根据目标代码段 descriptor 定义的两种情况：

1）、nonconforming target code segment

★ 直接转移后的权限级别是不能必改变的。因此，**CPL** 必须要等于目标代码段的 **DPL**。

★ 要有足够的请求权限进行访问。因此，目标代码段选择子的 **RPL** <= **CPL**

2）、conforming target code segment

★ conforming code segment 允许访问高权限级别的代码。这里只需检查 **CPL** >= **DPL** 即可，**RPL** 忽略不检查。

★ 转移后 **CPL** 不会改变。

2、 以上两步通过后，处理器加载目标代码段的 CS 进入 CS register，但权限级别不改变，继而 **RPL** 被忽略。

★ 处理器根据 CS selector 在相应的 descriptor table 找到 code segment descriptor。CS 的 Bit2（TI 域） 指示在哪个 descriptor table 表查找，CS.TI = 0 时在 GDT 查找，CS.TI = 1 时在 LDT 查找。

★ CS 的 Bit15~Bit3 是 selector index 值，处理器基于 GDT 或 LDT 来查找 segment descriptor。具体是：GDTR.base 或 LDTR.base + CS.SI × 8 得出 code segment descritpro。

★ 处理器自动加载 code segment descriptor 的 base address、segment limit 及 attribute 域进入 CS register 的相应的隐藏域。

★ 转到 CS.base + eip 处执行指令

总结：用代码形式来说明直接转移 call cs:eip 这条指令

例： call 1a:804304c （即 cs = 1a, eip = 804304c）

```
target_cs = 1a;
target_eip = 0x0804304c;
CPL = CS.RPL;              /* 当前执行的代码段的权限级别就是 CPL */
RPL = target_cs.RPL;       /* 目标段 selector 的低 3位是 RPL */
target_si = target_cs.SI;     /* 目标段 selector 的索引是 Bit15~Bit3 */
```

```
target_ti = target_cs.TI;     /* 目标段 selector的描述符表索引是 Bit2 */
CODESEG_DESCRIPTOR target_descriptor;

if (target_ti == 0) { /* target_cs.TI为 0就是参考到 GDT(全局描述符表)  */
/* 以 GDTR寄存器的 base为基地址加上 selector的索引乘以 8即得出目标
       段描述符，目标描述符的 DPL就是目标段所需的访问权限  */
    target_descriptor = GDTR.base + target_si * 8

} else {                /* 否则就是参考 LDT (局部描述符表) */
    /* 以 LDTR寄存器的 base为基地址得出目标段描述符  */

target_descriptor = LDTR.base + target_si * 8;
}
DPL = target_descriptor.DPL;      /* 获取 DPL */
if (target_descriptor.type & 0x06) { /* conforming */
if (CPL >= DPL) {    /* 允许执行高权限代码  */
     /* go ahead */
    } else {
     /* 引发 #GP异常  */
    goto DO_GP_EXCEPTION;
}

} else {                    /* nonconforming */
    if (CPL == DPL && RPL <= CPL) {
        /* go ahead */
    } else {
        /* 引发 #GP异常  */
       goto DO_GP_EXCEPTION;
    }

}
/****** go ahead … … ******/
CS = target_cs;             /* 加载目标段 CS进入 CS寄存器  */
EIP = target_eip;          /* 加载目标指令 EIP进入 EIP寄存器  */
                  /* 当前执行权限 CPL不改变  */

goto target_descriptor.base + target_eip; /* 跳转到目标地址执行  */

DO_GP_EXCEPTION:      /* 执行 #GP异常点  */
          … …
```

二、 使用 call gate 进行控制权的转移

使用 call gate 进行转移控制，目的是建立一个利用 gate 进行向高权限代码转移的一种保护机制。gate 符相当一个进入高权限代码的一个通道。

对于指令 call cs:eip 来说：
★ 目标代码的 selector 是一个门符的选择子。用来获取门描述符。
★ 门描述符含目标代码段的 selector 及目标代码的偏移量。
★ 目标代码的 ip 值被忽略。因为门符已经提供了目标代码的偏移量。

1、 权限的检查
★ 首先，必须要有足够的权限来访问 gate 符，所以：CPL <= DPLg（门符的 DPL）且：RPL <= DPLg
★ 进前代码向高权限代码转移,所以,对于 conforming 类型的代码段来说,必须 CPL >= DPLs（目标代码段的 DPL）
★ 对于 nonconforming 类型代码段来说，必须 CPL = DPLs
★ call 指令改变当前权限，而 jmp 指令不改变当前权限。

总结:

```
if ((CPL <= DPLg) && (RPL <= DPLg)) { /* 足够的权限访问门符 */
if (target.C == CONFORMING) {   /* 目标代码属于 conforming类型 */
    /* 向高权限级别代码转移控制 */
    if (CPL >= DPLs) {
          /* 通过访问 */
      } else {
           /* 失败， #Gp异常发生 */
    }

    } else {                /* 目标代码属于 nonconforming 类型 */
       /* 平级转移 */
    if (CPL == DPLs) {
          /* 通过访问 */
        } else {
           /* 失败， #GP 异常发生 */
        }
}

} else {   /* 没有足够权限访问门符 */
/* #GP 异常发生 */
}
```

2、 控制权的转移
指令：call 1a:804304c （其中 1a 是 call gate selector）

```
gate_selector = 0x1a;                  /* call gate selector */
RPL = gate_selector.RPL;               /* 门符选择子 RPL */
CPL = CS.RPL;                         /* 当前代码段低 3位是 CPL*/
CALLGATE_DESCRIPTOR call_gate_descriptor;       /* 门符的描述符 */
CODESEG_DESCRIPTOR target_cs_descritpor;       /* 目标代码段的描述符 */
```

```
call_gate_si = gate_selector.SI;       /* 门符 selector 的索引 */
call_gate_ti = gate_selector.TI;       /* 门符 selector的描述符表索引 */

/* 获取 call gate descriptor */
if (call_gate_ti == 0) {       /* TI为 0 就是参考到 GDT (全局描述符表)  */
/* 以 GDTR寄存器的 base 为基地址加上 selector的索引乘以 8即得出门符的描述符,
门符的 DPL就是门符的访问权限 */
    call_gate_descriptor = GDTR.base + call_gate_si * 8;
} else {                /* 否则就是参考 LDT (局部描述符表) */
     /* 以 LDTR寄存器的 base 为基地址得出目标段描述符 */
call_gate_descriptor = LDTR.base + call_gate_si * 8;
}

/* 获取 target code segment descriptor */
target_cs = call_gate_descriptor.selector;    /* 获取门符的目标代码段选择子
*/
target_cs_si = target_cs.SI;          /* 目标代码段的索引 */
target_cs_ti = target_cs.TI;                  /* 目标代码描述符表索引 */
if (target_cs_ti == 0)
    target_cs_descriptor = GDTR.base + target_cs_si * 8;
else
    target_cs_descriptor = LDTR.base + target_cs_si * 8;

DPLg = call_gate_descriptor.DPL;      /* 获取门符的 DPL */
DPLs = target_cs_descriptor.DPL;      /* 获取目标代码段的 DPL */

if (CPL <= DPLg && RPL <= DPLg) {   /* 有足够权限访问门符 */
     if (target_cs_descriptor.type & 0x06) { /* conforming */
          if (CPL >= DPLs) {
                /* 允许访问目标代码段 */
          } else {
                /* #GP 异常产生 */
          }
     } else if (CPL == DPLs) { /* nonconforming */
         /* 允许访问目标代码段 */
     } else {
          /* 拒绝访问, #GP 异常发生 */
          goto DO_GP_EXCEPTION;
     }
} else { /* 无权限访问门符 */
     /* 拒绝访问,  #GP异常发生 */
     goto DO_GP_EXCEPTION;
}
/* 允许访问 */
```

```
current_CS = target_cs;          /* 加载目标代码段进入 CS 寄存器 */
current_CS.RPL = DPLs;           /* 改变当前执行段权限 */
current_EIP = call_gate_descriptor.offset;   /* 加载 EIP */
/* 跳转到目标代码执行 */
/* goto current_CS:current_EIP */
goto target_cs_descriptor.base + call_gate_descriptor.offset;
return;
DO_GP_EXCEPTION:          /* 执行异常 */
```

三、 使用中断门或陷井门进行转移

中断门符及陷井门必须存放在 IDT 中，IDT 表也可以存放 call gate。

1、 中断调用时的权限检查

用中断门符进行转移时,所作的权限检查同call gate相同,区别在于intterrupt gate 转移不需要检查 RPL，因为，没有 RPL 需要检查。

★ 必须有足够的权限访问门符，CPL <= DPLg

★ 向同级权限代码转移时，CPL == DPLs，向高权限代码转移时，CPL > DPLs

总结

```
if (CPL <= DPLg) { /* 有足够权限访问门符 */
    if (CPL >= DPLs) {
        /* 允许访问目标代码头 */
    } else {
         /* 失败， #GP异常发生 */
    }

} else {
/* 失败， #GP异常发生 */
}
```

2、 控制权的转移

发生异常或中断调用时

★ 用中断向量在中断描述符表查找描述符：中断向量×8，然后加上 IDT 表基址得出描述符表。

★ 从查找到的描述符中得到目标代码段选择子，并在相应的 GDT 或 LDT 中获取目标代码段描述符。

★ 目标代码段描述符的基址加上门符中的 offset，确定最终执行入口点。

中断或陷井门符转移的总结：

例： int 0x80 指令发生的情况

```
vector = 0x80;
INTGATE_DESCRIPTOR gate_descriptor = IDTR.base + vector * 8;
CODESEG_DESCRIPTOR target_descriptor;
```

```
TSS tss = TR.base;                  /* 得到 TSS 内存块 */
DPLg = gate_descriptor.DPL;
target_cs = gate_descriptor.selector;
if (CPL <= DPLg) {                  /* 允许访问门符 */

if (target_cs.TI == 0) {    /* index on GDT */
    target_descriptor = GDTR.base + target_cs.SI * 8;
} else {                /* index on LDT */
target_descriptor = LDTR.base + target_cs.SI * 8;
    }

DPLs = target_descriptor.DPL;


if (CPL > DPLs) {       /* 向高权限代码转移 */

    /* 根据目标代码段的 DPL值来选取相应权限的 stack结构 */
    switch (DPLs) {
    case 0 :      /* 假如目标代码处理 0级，则选 0级的 stack结构 */
             SS = tss.ss0;
             ESP = tss.esp0;
             break;
        case 1:
             SS = tss.ss1;
            ESP = tss.esp1;
             break;
        case 2:
             SS = tss.ss2;
             ESP = tss.esp2;
             break;
    }

       /* 以下必须保护旧的 stack结构，以便返回 */
    *--esp = SS;            /* 将当前 SS入栈保护 */
    *--esp = ESP;           /* 将当前 ESP入栈保护 */

} else if (CPL == DPLs) {
     /* 同级转移，继续向下执行 */
} else {
    /* 失败，# GP异常产生，转去处理异常 */
}


*--esp = EFLAGS;            /* eflags 寄存器入栈 */
```

```
    /* 分别将 NT、 NT、 RF及 VM标志位清 0 */
EFLAGS.TF = 0;
EFLAGS.NT = 0;
EFLAGS.RF = 0;
EFLAGS.VM = 0;

if (gate_descriptor.type == I_GATE32) { /* 假如是中断门符 */
EFLAGS.IF = 0;          /* 也将 IF标志位清 0, 屏蔽响应中断 */
    }

    *--esp = CS;              /* 当前段选择子入栈 */
    *--esp = EIP;             /* 当前 EIP入栈 */
   CS = target_selector;      /* 加载目标代码段 */
CS.RPL = DPLs;            /* 改变当前执行权限级别 */
EIP = gate_descriptor.offset; /* 加载进入 EIP */

/* 执行中断例程 */
goto target_descritptor.base + gate_descriptor.offset;

} else {
/* 失败, #GP 异常产生，转去处理异常 */
}
```

■ 堆栈的切换

控制权发生转移后，处理器自动进行相应的堆栈切换。
1、 当转向到同权限级别的代码时，不会进行堆栈级别的调整，也就是不进行堆栈切换。
2、 当转向高权限级别时，将发生相应级别的堆栈切换。从 TSS 块获取相应级别的 stack 结构。

例：假如当前运行级别 CPL 为 2 时，发生了向 0 级代码转移时：

```
TSS tss = TR.base;          /* 从 TR寄存器中获取 TSS 块 */
CPL = 2;                  /* 当前运行级别为 2 级 */
DPL = 0;                  /* 目标代码需要级别为 0 级 */

/* 根据目标代码需要的级别进行选取相应的权限级别的 stack结构 */
switch (DPL) {
case 0:
    SS = tss.ss0;
    ESP = tss.esp0;
    break;
case 1:
```

```
    SS = tss.ss1;
    ESP = tss.esp1;
    break;
case 2:
    SS = tss.ss2;
    ESP = tss.esp2;
    break;
}
*--esp = SS;          /* 保存旧的 stack结构 */
*--esp = ESP;
/* 然后再作相当的保存工作，如保存参数等 */
*--esp = CS;          /* 最后保存返回地址 */
*--esp = EIP;
```

■ 控制权的返回

当目标代码执行完毕，需要返回控制权给原代码时，将产生返回控制权行为。返回控制权行为，比转移控制权行为简单得多。因为，一切条件已经在交出控制权之前准备完毕，返回时仅需出栈就行了。

1、 near call 的返回
近调用情况下，段不改变，即 CS 不改变，权限级别不改变。仅需从栈中 pop 返回地址就可以了。

2、 直接控制权转移的返回（far call 或 far jmp）
直接控制权的转移是一种不改变当前运行级别的行为。只是发生跨段的转移。这时，CS 被从栈中 pop 出来的 CS 值加载进去，处理器会检查 CPL 与这个 pop 出来的选择子中的 RPL 进行检查，相符则返回。不相符则发生 #GP 异常。

总结：假如当前运行的目标代码执行完毕后，将要返回。这时 CPL 为 2

```
CPL = 2;          /* 当前代码运行级别为 2 */
... ...
EIP = *esp++;     /* pop出原 EIP 值 */
CS = *esp++;      /* pop出原 CS值 */
if (CPL == CS.RPL) {
/* CS.RPL 代表是原来的运行级别，与 CPL相符则返回 */
return ;
} else {
/* #GP异常产生，执行异常处理 */
}
```

3、 利用各种门符进行向高权限代码转移后的返回
从高权限代码返回低权限代码，须从 stack 中 pop 出原来的 stack 结构。这个 stack 结构

属于低权限代码的 stack 结构。然后直接 pop 出原返回地址就可以了。

总结：

```
CPL = 0;            /* 当前运行级别为 0 级 */
... ...
EIP = *esp++;       /* 恢复原地址 */
CS = *esp++;        /* 恢复原地址及运行级别 */

ESP = *esp ++;        /* 恢复原 stack结构 */
SS = *esp++;        /* 恢复原 stack 结构，同时恢复了原 stack访问级别 */
return ;          /* 返回 */
```