第一次代码阅读报告

1600011338　　蒲伟良

1.请简述xv6系统的启动过程，并与上课所讲授的内容进行对比。

1. 执行BIOS，BIOS会进行硬件的准备工作，之后BIOS会将从引导扇区中加载引导加载器即bootloader的代码，然后将控制权交给bootloader
2. xv6中bootloader从`bootasm.S`的`start`开始，主要进行以下操作

- 屏蔽中断
- 设置 ds、es、ss三个寄存器为0
- 通过设置键盘A20端口，启用高位地址（20位以上）
- 设置cr0的PE位启动保护模式，并通过`jmp`指令进入保护模式
- 从`start32`开始后设置ds、es、ss为`0x10`，即内核将要加载到的地方`0x10000`，初始化fs和gs
- 调用`bootmain()`，bootloader的主体，在`bootmain.c`中，后面的初始化操作见第2题。

3. xv6启动过程和课上说的一样，没什么差别，只不过从代码上可以看到更多的细节，比如通过A20端口来禁止或者开启20位以上的地址

2. 请简述bootmain()函数的内容，并结合其代码谈谈bootloader在xv6系统启动中起到了什么作用？boot loader是如何决定该加载多少个扇区以至于能加载整个内核？它是从哪里得到这一信息的？

bootloader主要在`bootasm.S`，`bootmain.c`，`entry.S`中，第1题可以看到代码跳转到了`bootmain()`中，然后

```
//in bootmain.c
elf = (struct elfhdr*)0x10000;
readseg((uchar*)elf, 4096, 0);
```

- 从磁盘中加载内核到内存地址`0x10000`，加载4096字节，内核文件是以ELF格式编写的（ELF头在`elf.h`中定义，同时还定义了程序头），由于已知ELF头的大小，所以加载4096字节是安全的，保证头部已经全部读取了。
- `if(elf->magic != ELF_MAGIC) return;`根据读取的ELF头判断魔数看是不是ELF文件

```
//in bootmain.c
ph = (struct proghdr*)((uchar*)elf + elf->phoff);
eph = ph + elf->phnum;
for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
        stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
}
```

- 从上面的代码段可以看出接下来根据ELF头中的信息，即program section的数量和每个section的起点读取剩下的内核文件到内存中，并线性地保存。即bootloader从ELF头中的信息得知要加载的扇区数量和偏移。

```
//in bootmain.c
entry = (void(*)(void))(elf->entry);
entry();
```

- 上面的代码通过ELF头部的入口进入`entry()`函数，该函数主要进行分页机制相关的操作，比如指定分页的目录地址，将页大小扩展到4MB，最后将调用`main()`
- bootloader顾名思义就是一个loader，主要作用就是将原本在磁盘中的xv6内核加载到内存当中。BIOS比较简单，只能加载较小，较简单的bootloader，然后bootloader在去加载功能更加多样的操作系统，当然加载之前还要做一些硬件上的准备，这个在第1题已经提到了。总之，bootloader就是初始化一些硬件，准备好环境，将内核加载到内存中。

3. 什么是用户态和内核态？两者有何区别？什么是中断和系统调用？两者有何区别？计算机在运行时，是如何确定当前处于用户态还是内核态的？

- 内核态是运行操作系统的态，具有最高的权限，可以调用只有内核态下才能调用的函数。用户态是用户运行程序的态，权限较低，不能调用某些接口和函数。
- 用户态和内核态的主要区别是权限级别不同，在intel的x86架构的CPU中，保护模式下，需要从cs寄存器取得地址的一部分，而cs寄存器的后两位即CPL位明确了执行指令时的权限，0代表最高，3最低，一般0是内核态，而1、2、3可能是不同的权限用户，xv6中只有0或者3，3为用户态。
- 中断包括了来自硬件的中断（比如时钟）和软件执行过程中的异常（比如除零），而系统调用是应用程序主动向操作系统请求调用操作系统提供的接口函数（比如`fork()`）
- 中断是有硬件产生的或者是指令执行异常导致的，对于CPU来说是无法预测的，然而系统调用是带有目的性的，是应用程序的主动请求，是可预测的，说白了只是函数调用，只不过因为权限的限制需要陷入内核。

4. 计算机开始运行阶段就有中断吗？xv6的中断管理是如何初始化的？xv6 是如何实现内核态到用户态的转变的？XV6 中的硬件中断是如何开关的？实际的计算机里，中断有哪几种？

- BIOS运行支持部分中断，所以一开始就有中断。
- xv6进入`main()`后，从注释可知`lapicinit()`和`ioapicinit()`初始化中断控制器，然后`tvinit()`初始化中断描述符表，在这之前需要初始化好的中断向量表。
- 中断向量表记录在`trapvector.S`中，由`vector.pl`生成。中断向量表总计有256个向量，表中记录了中断触发后要执行的代码，下面给出了示例。都是先push一个或两个参数（只有少数几个只push一个参数，第一个参数都是0，代表errorcode，一部分中断CPU会自动push这个errorcode，所以不需要手动push，第二个参数是trapno，表示引发中断的编号），然后调用`trapasm.S`中的`alltrap()`，即初始化了中断后的代码。

```
//in trapvector.S (generated by vecotr.pl)
vector0:
    pushl $0
    pushl $0
    jmp alltraps
```

- `tvinit()`通过调用在`mmu.h`中定义的函数`SETGATE()`来中断描述符表，使得中断触发后CPU知道有足够的信息操作，中断描述符表的细节在第5题和第8题
- 随后`main()`调用`mpmain()`，`mpmain()`再调用`idtinit()`，将之前初始化好的中断向量表载入到相应的寄存器中保存，使得触发中断后CPU能够找到中断向量表。
- 内核态到用户态是靠`trapasm.S`中的`trapret`函数的最后一条指令`iret`实现的，`iret`指令会从中断帧trapframe中恢复之前保存的发现中断时的cs寄存器的值（trapframe保存在内核的栈中），前面已经提到了用户态和内核态标志性的区别是cs寄存器里的权限CPL，如果中断前在内核态，则重新回到内核态，否则就会回到用户态。

- 硬件中断的开关是汇编命令cli和sti，cli屏蔽课屏蔽中断，sti允许可屏蔽中断。
- xv6中用中断统称来自硬件的中断，程序出错的陷入和系统调用
- 实际的计算机里，中断可以按中断的原因和中断的功能分类。
    - 按原因：中断一般有trap，fault，abort和interrupt，一般为4种。trap是有意识安排的，程序员编写的，fault指可恢复的错误，能够继续运行程序，abort指不可恢复的错误，触发后只能停止或杀死进程，interru指来自硬件的中断。
    - 按功能：中断分为输入输出中断、外中断、机器故障中断、程序性中断、访管中断

5. 什么是中断描述符，中断描述符表？在 XV6 里是用什么数据结构表示的?

- 触发中断后，CPU需要知道中断的信息，比如是什么中断，中断后的处理程序，这些都在中断描述符中，中断描述符在xv6中定义在`mmu.h`中，如下

```
//in mmu.h
struct gatedesc {
  uint off_15_0 : 16;   // low 16 bits of offset in segment
  uint cs : 16;         // code segment selector
  uint args : 5;        // # args, 0 for interrupt/trap gates
  uint rsv1 : 3;        // reserved(should be zero I guess)
  uint type : 4;        // type(STS_{IG32,TG32})
  uint s : 1;           // must be 0 (system)
  uint dpl : 2;         // descriptor(meaning new) privilege level
  uint p : 1;           // Present
  uint off_31_16 : 16;  // high bits of offset in segment
};

//in trap.c
struct gatedesc idt[256];
```

- 中断描述符提供了与中断号相关的中断向量表的偏移，cs寄存器的值，门的类型（陷阱/中断）、触发该中断需要的优先级DPL，通过这些信息CPU可以知道中断后执行什么程序，处理完之后如何返回原来的态
- 中断描述符表在`trap.c`中定义，见上面的代码，总共有256项，xv6规定了0到31为trap，32到63为interrup，64为系统调用

6. 请以某一个中断（如除零，页错误等）为例，详细描述 XV6 一次中断的处理过程。包括： 涉及哪些文件的代码？如何跳转？内核态，用户态如何变化？涉及哪些数据结构等等。

- 这里以除零的中断为例，除零会引发trapno为0的中断。
- 中断发生后，CPU根据目前的状态保存相应的寄存器，如果处于用户态，会首先压入%ss和%esp，保存用户栈，然后压入eflags、%cs、%eip、可能有的errorcode。内核态因为不涉及栈的改变，不会压入%ss和%esp，但其他相同。
- CPU根据trapno查中断描述符表，即`trap.c`中的`idt[]`，根据表中相应的中断描述符，跳转到相应的入口`vector[]`，继续压入参数，跳转到`trapasm.S`中的`alltrap`中
- `alltrap`会继续压入%ds、%es、%fs、%gs调用`pushal`将常规的寄存器都压入栈，包括%eax、%ecx、%edx、%ebx、%osep、%ebp、%esi、%edi，然后设置%ds和%es指向内核，并压入%esp作为参数，最后调用`trap(tf)`，tf表示中断帧，储存在压入的%esp中。前面的一系列操作其实压入了一个中断帧，中断帧在`x86.h`中定义，代码如下（由于栈是向地址小的方向生长的，所以压入顺序和定义的相反）。

```
// in x86.h
struct trapframe {
```

```
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort gs;
  ushort padding1;
  ushort fs;
  ushort padding2;
  ushort es;
  ushort padding3;
  ushort ds;
  ushort padding4;
  uint trapno;

  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding5;
  uint eflags;

  // below here only when crossing rings, such as from user to kernel
  uint esp;
  ushort ss;
  ushort padding6;
};
```

- `trap()` 在 `trap.c` 中定义，该函数对于输入的中断帧，检查中断号trapno，如果是系统调用，就调用 `syscall()`，系统调用的trapno是64，除零不是，接着检查是不是大于31的interr，如果是就分别处理，发现也不是，进入default，首先检查发生中断时是不是在内核态（`tf->cs` 记录了当时的状态），如果是，说明是操作系统的锅，输出一个错误，如果不是，告诉用户是你自己的错。
- 中断是唯一的从用户态进入内核态的方式，而从内核态返回比较容易，前面也提到了是 `iret` 后根据保存在中断帧中的cs寄存器恢复的（该寄存器由CPU自动压入栈）。而从用户态进入内核态比较复杂，CPU会从中断描述符中取出cs的值替换掉原来的，新的寄存器值CPL为0，然后跳转后进入内核态。

7. 请以系统调用 setrlimit（该系统调用的作用是设置资源使用限制）为例，叙述如何在 XV6 中实现一个系统调用。（提示：需要添加系统调用号，系统调用函数，用户接口等等）。

- 系统调用号在 `syscall.h` 中定义，现在已有21个，不妨定义 `#define SYS_setrlimit 22` 即定义为22号系统调用
- 用户接口在 `syscall.c` 中定义，首先要声明外部函数 `extern sys_setrlimit`，说明该系统调用函数实现来自其他文件，然后在 `syscalls[]` 中添加入口 `[SYS_setrlimit] sys_setrlimit`
- 系统调用函数可以在任意文件中实现，只要 `syscall.c` 中有 `#include`，但不难发现一般都会写在 `sysproc.c` 和 `sysfile.c` 中。

8. 其他你认为有趣有价值的问题。

保护模式和实模式

- BIOS转到bootloader后，处于实模式，需要转换到保护模式下工作。实模式下的寻址方式为物理地址 = 段基地址 + 段内偏移地址，可见实模式能够比较自由地寻址，用户可以轻易地修改每一个物理地址，而保护模式对此进行了一定的隔离，避免没有权限的用户自由修改地址内的内容。
- 保护模式下使用分段的方式寻址，一开始的地址是逻辑地址，表示为段选择符加偏移地址，段选择子保存在段选择子寄存器中，即前面提到的%cs、%ds、%es、%ss等，分别用于选择代码、数据和栈，段选择符是16位的寄存器(见下图)，0:1为特权级，对于cs来说是当前的特权CPL（即区分用户态和内核态），对于其它segment selector来说是用户请求的特权级RPL，2位是table indicator，用来区分GDT和LDT，GDT和LDT就是两个储存段描述符的表（GDT能够在任意装态下可见），3:15位是index，用来在GDT或者LDT中找到相应的段描述符。

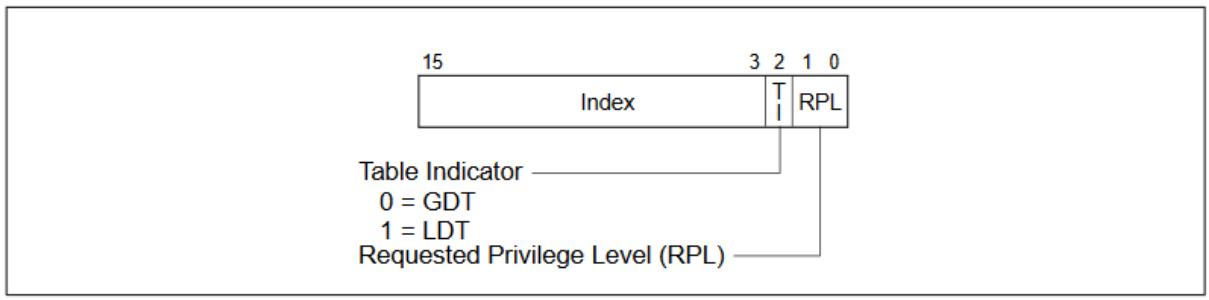


Figure 3-6. Segment Selector

图片来源Intel® 64 and IA-32 Architectures Software Developer Manuals

- 段描述符用于描述段（见下图），通过base address可以寻址LIMIT表示段的大小，DPL是描述符权限。

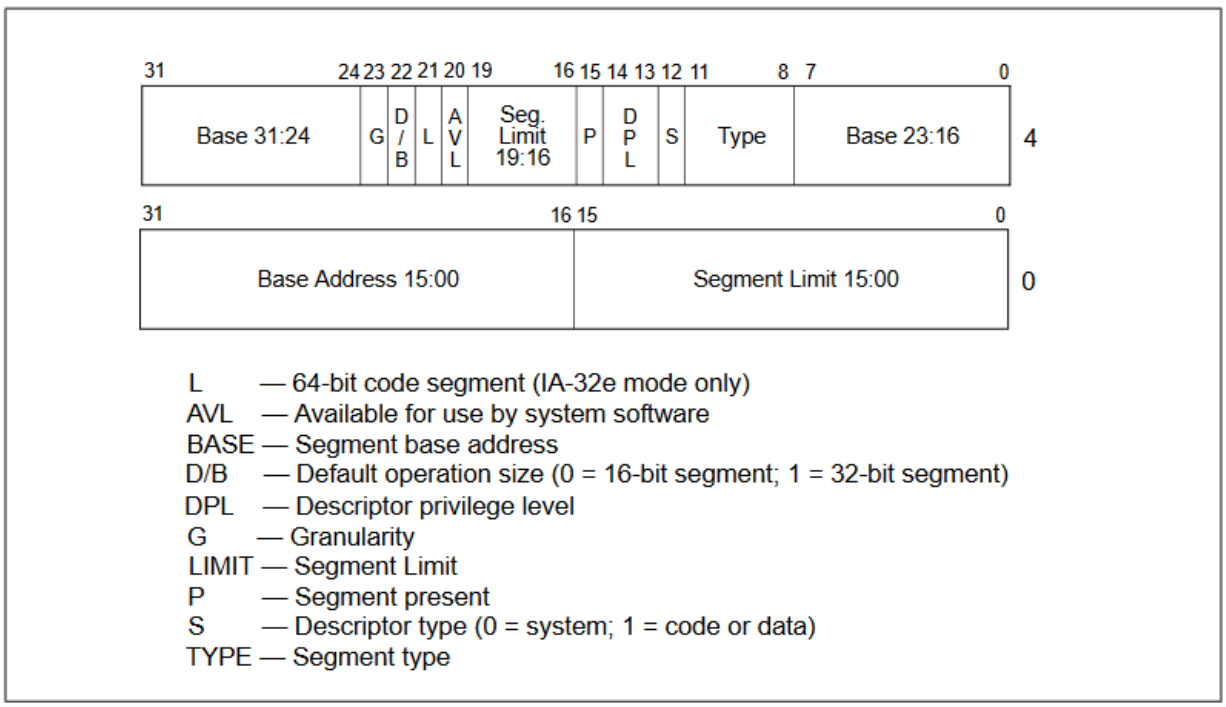


Figure 3-8. Segment Descriptor

图片来源Intel® 64 and IA-32 Architectures Software Developer Manuals

- base address和偏移地址相加得到了线性地址，如果没有开启分页模式则线性地址代表了物理地址。

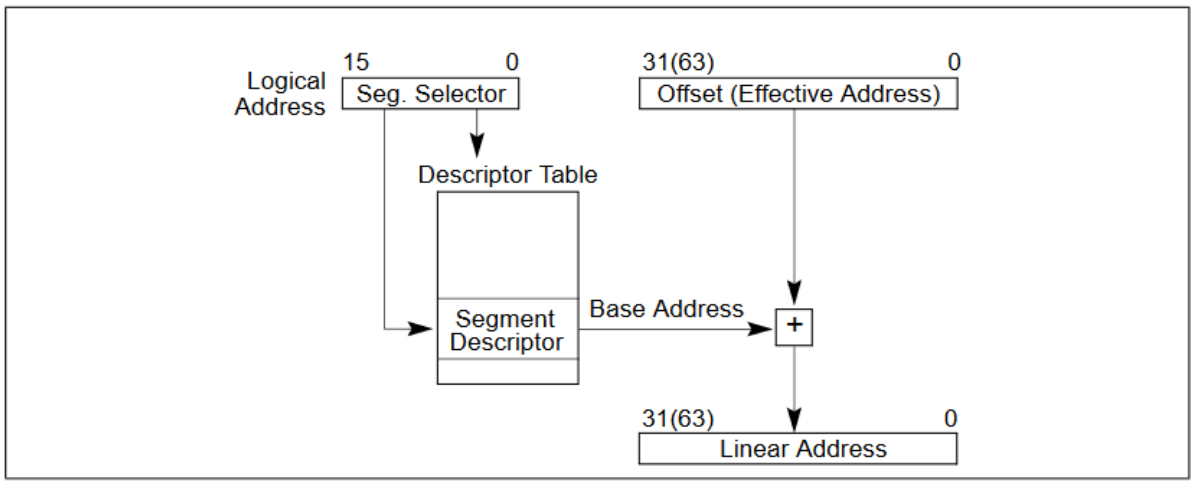
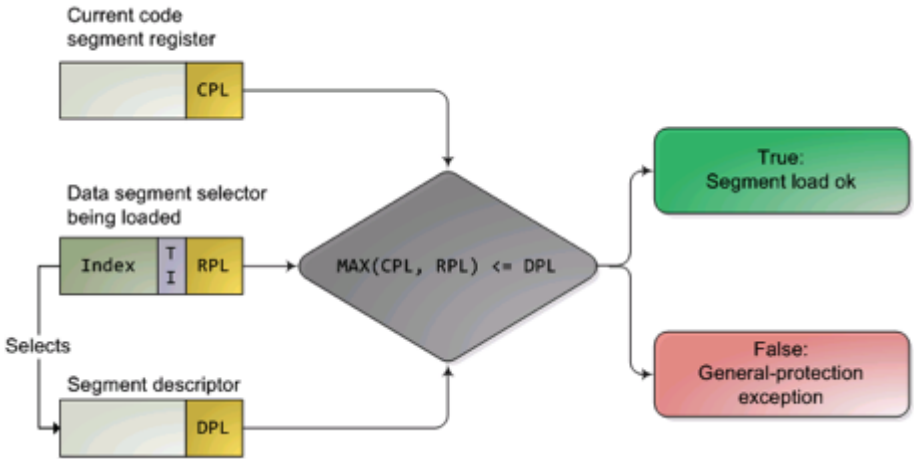


Figure 3-5. Logical Address to Linear Address Translation

图片来源Intel® 64 and IA-32 Architectures Software Developer Manuals

- CPU在这个过程会比较CPL、DPL和RPL（如下图），CPL<=DPL保证了用户不会越权，而RPL是由程序员设置的，算是程序员编写过程加的保护。



图片来源ucore实验指导书

打开保护模式

```
# in bootasm.S
lgdt    gdtdesc
movl    %cr0, %eax
orl     $CR0_PE, %eax
movl    %eax, %cr0

# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
  SEG_NULLASM                             # null seg
  SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
  SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg

gdtdesc:
  .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
  .long   gdt                             # address gdt
```

- 上面的代码为`bootasm.S`中打开保护模式，首先载入GDT的描述符`gdtsesc`，从这里可以看到GDT表中，第0段是空的，不使用，第1段是内核的代码段，第2段是内核数据段，所以不难理解`mmu.h`中定义的`#define SEG_KCODE 1 // kernel code`和`#define SEG_KDATA 2 // kernel data+stack`。接下来设置%cr0的PE位，该位控制保护模式的开关。%cr0主要用于控制处理器操作模式和状态的系统控制标志。（见下图）

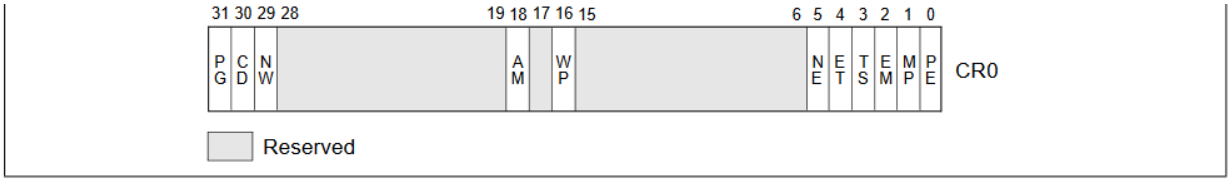


Figure 2-7. Control Registers

图片来源Intel® 64 and IA-32 Architectures Software Developer Manuals

中断的一些细节

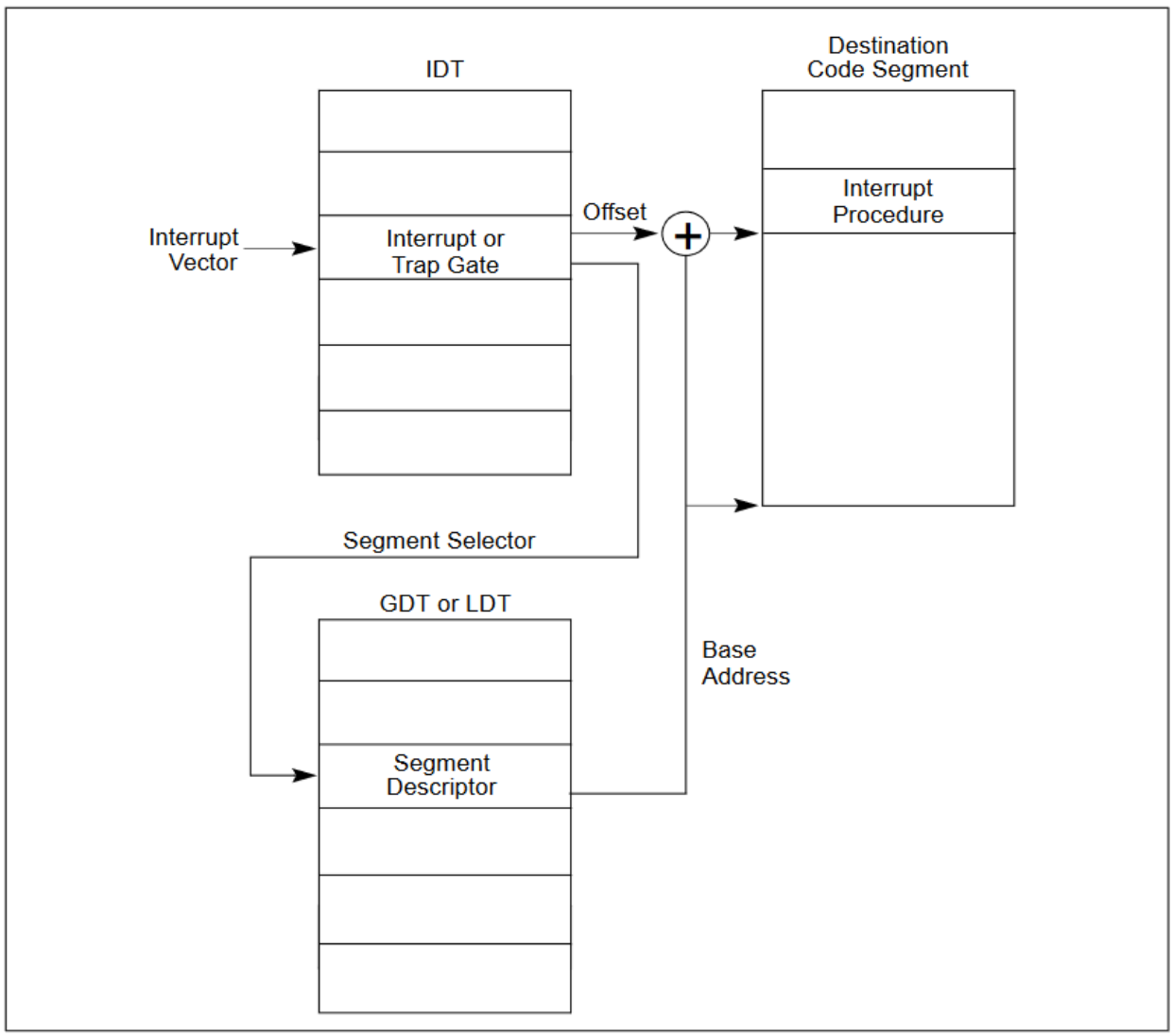


Figure 6-3. Interrupt Procedure Call

图片来源Intel® 64 and IA-32 Architectures Software Developer Manuals

- 从上面的图片可知中断到来后，CPU查找中断描述符表IDT，索引到相应的中断描述符，第5题中已经给出中断描述符的结构。中断描述符中有段选择子和偏移地址，构成了逻辑地址，CPU据此生成线性地址（即在GDT表

中索引相应的段描述符，取出段基地址，和偏移相加）。`trap.c`中用下面的代码初始化中断描述符表。首先对每个中断描述符设置段选择子为0x1000，即GDT表中第1个段，为内核的代码段，并且设置CPL为0，为内核态。这样当中断触发后，CPU就会从这里陷入内核态。然后设置偏移，即中断的代码的位置，最后设置DPL为0，限制只有内核态能进行中断代码的调用。不过系统调用可以被用户调用，所以设置了DPL为3。

```
// in trap.c
for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

// in mmu.h
#define SETGATE(gate, istrap, sel, off, d)                \
{                                                         \
  (gate).off_15_0 = (uint)(off) & 0xffff;                \
  (gate).cs = (sel);                                      \
  (gate).args = 0;                                        \
  (gate).rsv1 = 0;                                        \
  (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
  (gate).s = 0;                                           \
  (gate).dpl = (d);                                       \
  (gate).p = 1;                                           \
  (gate).off_31_16 = (uint)(off) >> 16;                  \
}
#endif
```

主要参考文档

1. [Intel® 64 and IA-32 Architectures Software Developer Manuals] [https://software.intel.com/sites/default/files/managed/a4/60/325384-sdm-vol-3abcd.pdf]
2. [xv6中文文档][https://legacy.gitbook.com/book/th0ar/xv6-chinese/details]
3. [ucore实验指导书][https://objectkuan.gitbooks.io/ucore-docs/content/]