实验一 bootloader 启动 ucoreos

第一节 x86 启动顺序

1. 理解 x86-32 平台的启动过程

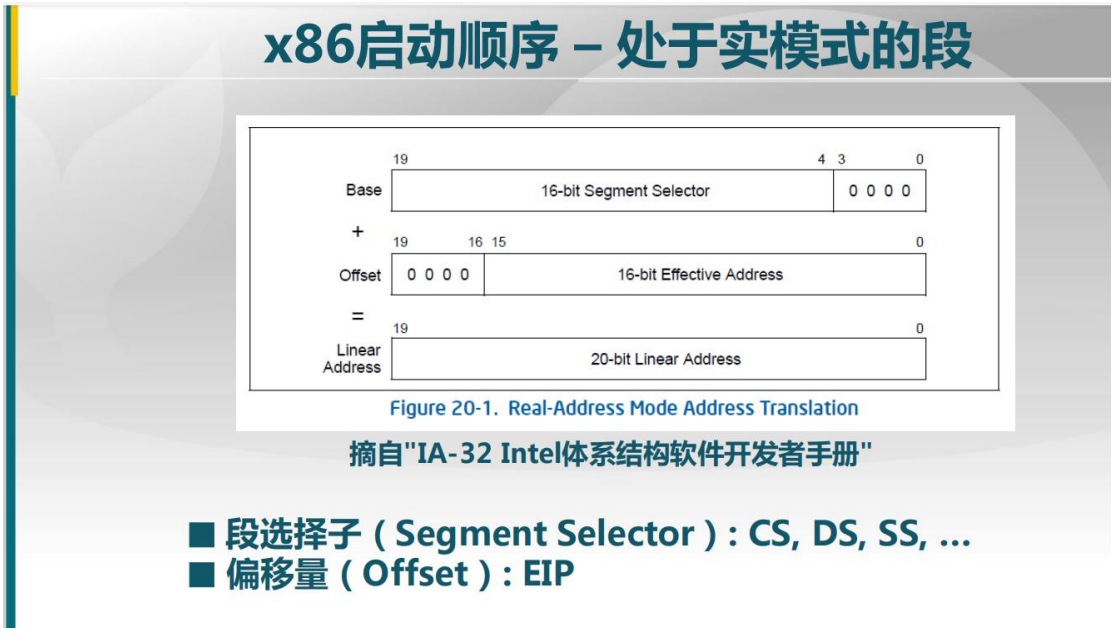
和 ucore 相关的寄存器有：段地址 CS 和 EIP 结合在一起来决定启动后的第一条地址，也就是形成启动之后，加电之后的一个地址，决定在哪个地址获得一条指令去执行。

加电后，初始状态，CS=F000H,EIP=0000FFF0H，需要注意，x86 一开始加电时，启动的是(16 位的)实模式(为了向下兼容 8086)，在这种情况下，寻址按照实模式的寻址方式，所以实际地址是：

Base+EIP,而 CS 的 Base 是多少呢？CS 是段寄存器，段寄存器中有一个隐含的叫做 Base 的内容，Base 代表基址，基址，其实就是存的值。Base+EIP=FFFF0000H+0000FFF0H=FFFFFFF0H. 因此 CS 的 Base 是 FFFF0000H。Base+EIP=FFFFFFF0H，这就是 BIOS 的 EPROM(Erasable Programmable Read Only Memory，可拭除且可程序只读存储器，一种可规划的只读存储器,在正常使用时只能进行读操作,但其中的讯息可用特殊的技术擦除并再编程)所在地，也就是加电之后要去取得那个内存所在的地址，这个内存是在 BIOS 中，也就是 PC 中的一个固件。注意上面这个地址是只读的一块地址。

从上面这个地址会得到第一条指令，这条指令一般是一条长跳转指令(这样 CS 和 EIP 都会更新)，会跳到 BIOS 中一个可以被访问的 1M 内存空间里面去执行(符合在实模式条件下，它的寻址空间只有 1M 的这个特征)。

当 CS 被新值加载，则地址转换规则将开始起作用。



上图，CS 是 16 位的一个段寄存器，再加上一个 offset,offset 就是 EIP，里面有 16 位地址，至于图上**为什么有 20 位地址**？是因为 16 位的段寄存器左移了 4 位。(这里 Base 应该就只取到了 F0000H)

假设 BIOS 完成了它的工作，也就是硬件初始化的工作，底层的硬件初始化工作，保证机器能够进行后续的正常工作，完成了很多各种外设、CPU 及内存的质检，完成这些之后，下一步：

BIOS 固件会加载存储设备（比如软盘、硬盘、光盘、USB 盘）上的第一个扇区(主引导扇区，Master Boot Record, or MBR) 的 512 字节到内存的 0x7c00，然后转跳到 @ 0x7c00 的第一条指令开始执行(IP 地址跳到 0x7c00)。这个扇区里面的代码是什么呢？Lab1 里的

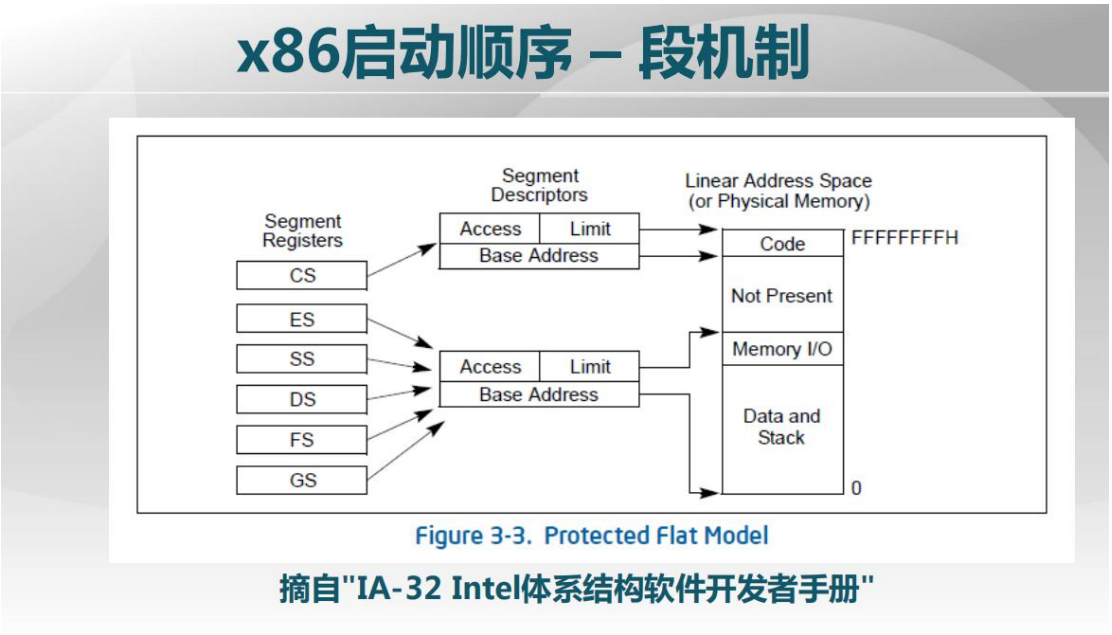
bootloader 就属于 512 字节这么一个特殊的执行代码。**它完成什么工作呢？**完成对 uCore 操作系统的进一步加载.问：**BIOS 为何不直接加载 uCore 操作系统？**答：取决于 BIOS 的能力问题，因为在刚开始设计时，BIOS 的功能就只是加载一个扇区，而一个操作系统的代码容量是大于 512 个字节，它会比较大，这种情况下，靠 BIOS 来负责加载一个复杂的容量很大的 OS 的话，不太现实，增加了 BIOS 工作的难度。因此 BIOS 只加载一个扇区，而这个扇区里的代码会完成后续的加载工作。这个扇区我们就称之为 Bootloader。

主引导扇区(0 号扇区)中的 Bootloader 代码完成的功能？

①从实模式切换到保护模式，从实模式 16 位的寻址空间切换到了 32 位的寻址空间，从 1M 的寻址切换到 4G 的寻址，为后续操作系统执行做准备：使能保护模式（protection mode）& 段机制（segment-level protection）(使能了保护模式，段机制也就自动加载上来正常工作了)

②从硬盘上读取 kernel in ELF 格式的 ucore kernel (跟在 MBR(主引导扇区)后面的扇区)并放到内存中固定位置

③跳转到 ucore OS 的入口点（entry point）执行，这时控制权到了 ucore OS 中(把 CS、EIP 的值指向操作系统内核所在内存中的起始点)



详细分析：

①从实模式切换到保护模式：

段机制：段寄存器 CS 起到了一个特殊的作用，即指针的作用，指向了一个段描述符(Segment Descriptors)，在段描述符里，描述了一个段落的起始地址和它的大小，所以可以根据 CS 中的 index 值，来找到 uCore 代码段的起始地址在什么地方以及大小。同理，数据段也可以用其它的一些特定的寄存器，段寄存器，比如 ES 或 DS 等等来表述，堆栈段 SS 等。但是，由于我们后面还有页机制，所以在段机制这一块，它的映射关系就尽量搞得简单(在 uCore 中，我们倾向于使用页机制来实现分段,这是 Lab2 涉及的内容)

但是这个段模式我们不能消掉，因为只要起了保护模式，在 X86 中，就规定这个段就 Enable 了，而且页机制是建立在段机制的基础之上实现的。

x86启动顺序 – 段机制

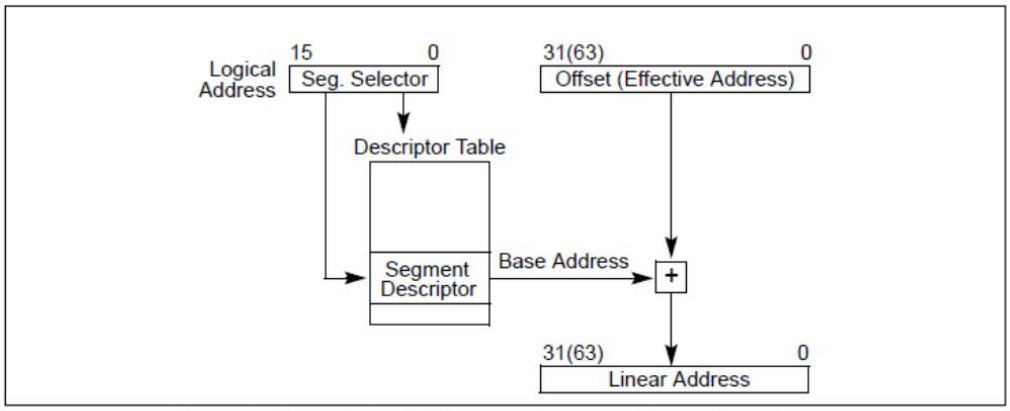


Figure 3-5. Logical Address to Linear Address Translation

摘自"IA-32 Intel体系结构软件开发者手册"

在一个段寄存器里面，会保存一块区域叫做段选择址(Seg. Selector),这个选择址就是我们刚才说的 index，它的值就代表 Index，这个 Index 会查找一个在段描述符表里面的一个项，叫做段描述符(Segment Descriptor)，段描述符里会存着刚才说的很重要的两个信息，也就是起始地址(基址，Base Address)和它的大小(Limit)，offset 就是我们说的 EIP(偏移量)，EIP 加上由 CS，或者说其它段寄存器所指出来的基址，叠加在一起，形成线性地址(Linear Address)。这里，由于没有启动页机制，所以线性地址就等同于物理地址。因此段机制就是一种映射关系。如果我们在段描述符中将 Base Address 设成零，意味着 EIP 的值也就是它的物理基址。

x86启动顺序 – 段机制

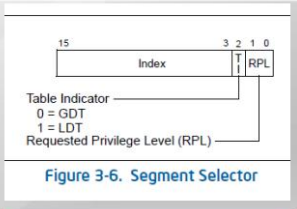


Figure 3-6. Segment Selector

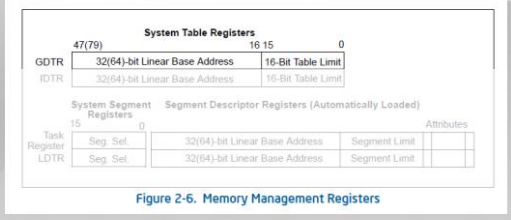


Figure 2-6. Memory Management Registers

```
Loading GDT:

lgdt gdtdesc

gdt:
     ......

gdtdesc:
     .word 0x17
     .long gdt
```

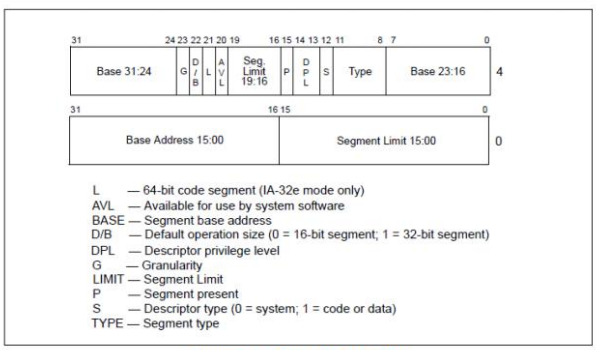


Figure 3-8. Segment Descriptor

摘自"IA-32 Intel体系结构软件开发者手册"

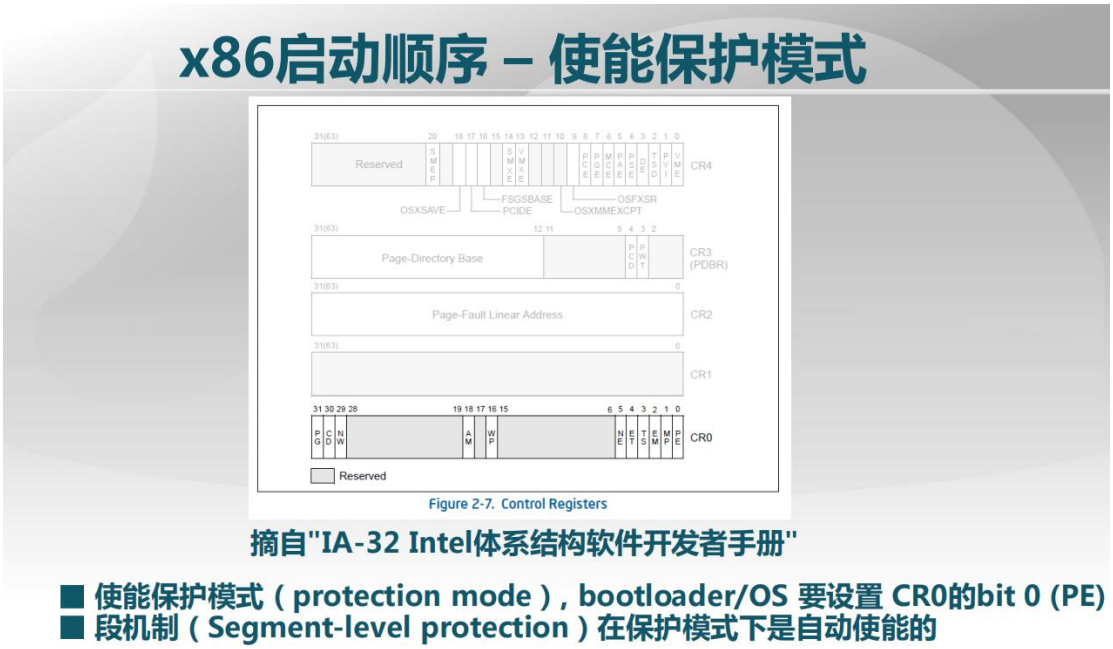
全局描述符表(GDT,也称段表)将各个段描述符装进去，它是由我们的 bootloader 来建立

的，通过 gdtdesc,可以使得 CPU 能够找到段表的起始地址，并通过内部的 GDTR 寄存器来保存相应的地址信息，使得各个 CS、DS、SS 等等可以和我们的 GDT 表建立对应关系。

以上信息很多，但我们主要关注两项：第一就是基址(Base)在什么地方，第二就是段的长度(limit)多大，其实在 uCore 中这个功能被弱化了，基址都是 0，段的长度都是 4G.

问题：各个段寄存器怎么产生 index?

答：段寄存器有 16 位，其中高 13 位放的就是 GDT 的 Index,接下来有两位，放的是 RPL，表明当前段的优先级的级别，即在 X86 中，用两位来表明优先级，意味着它可以表示 0 1 2 3 这四个特权级。一般说我们操作系统放在最高的是 0 特权级，应用程序放在 3 这个特权级里面。因此上图，段寄存器里面的 RPL 标记，表明当前执行时 CS 所处的特权级。TI 一般设为 0，因为我们用的是 GDT(全局描述符表),我们没有用到 LDT(本地描述符表)



最后，靠对一个特定的寄存器，即：系统寄存器(控制寄存器)CR0，把它的第 0 号 bit 置成 1，CPU 就进入保护模式。

总结：建好 GDT,GDT 的每一项是一个段描述符，要把相应段的段寄存器，CS、DS 等等设置好对应的 index,使得 CS、DS 等这些段寄存器能够指向全局描述符表 GDT 对应的项，这个项我们称之为段描述符，这个描述符指出映射关系，从而可以在使能了保护机制后，使得段机制能够正常的工作。

x86启动顺序 – 加载 ELF格式的ucore OS kernel

```
struct elfhdr {
  uint magic;       // must equal ELF_MAGIC
  uchar elf[12];
  ushort type;
  ushort machine;
  uint version;
  uint entry;       // program entry point (in va)
  uint phoff;       // offset of the program header tables
  uint shoff;
  uint flags;
  ushort ehsize;
  ushort phentsize;
  ushort phnum;      // number of program header tables
  ushort shentsize;
  ushort shnum;
  ushort shstrndx;
};
```

②加载 uCore OS：从硬盘上读取 kernel in ELF 格式的 ucore kernel (跟在 MBR(主引导扇区)后面的扇区)并放到内存中固定位置.uCore OS 编译出来会生成 elf 格式的执行程序，elf 格式是 Linux 中很常用的一种执行文件的格式，kernel 文件的内部信息使得 bootloader 可以根据这个文件的格式把 uCore 相应的代码、数据放到内存中相应的地址。Bootloader 需要去解析 elf 格式的信息，上图就是一个 elf 头，ELFheader 里指出了一个 Program Header，即程序段的头，程序段里包含了代码段，数据段等等。

上图中，unit phoff 就相当于 ELFHeader 的偏移地址(起始地址在什么地方)，以及 ushort PHnum，即 Program Header 的个数。

接下来，便可以进一步查找 Program Header 这个结构：

x86启动顺序 – 加载 ELF格式的ucore OS kernel

```
struct proghdr {
  uint type;        // segment type
  uint offset;      // beginning of the segment in the file
  uint va;          // where this segment should be placed at
  uint pa;
  uint filesz;
  uint memsz;       // size of the segment in byte
  uint flags;
  uint align;
};
```

上面 unit va 是虚地址，因为编出来的代码是要在某一个特定的地址上运行的，它指出了那个地址。unit memsz 是 Program Header 代码段的大小，这两个信息可以便于把内存中的相应一块区域用来存放 uCore 的代码段或者数据段。unit offset 指明从文件哪个位置把代码段或数据段读进来。

总结：这个流程也就是识别出一些很重要的关键信息，然后把相应的代码段、数据段从

我们的文件读到我们的内存中来。

以上，我们没有讲到文件系统，文件系统读的是最原始的磁盘扇区，把刚才 bootloader 之后的连续的磁盘扇区，连续读了 4 个磁盘扇区，读到内存中来，然后进行相应的分析工作。但要注意，随着后面 ucore size 的不断增加，后面可能读不止 4 个扇区，可能会读更多的扇区。

2. 理解 x86-32 的实模式、保护模式
3. 理解段机制

通过以上分析，已经介绍了实模式、保护模式、段机制。

第二节 C 函数调用的实现

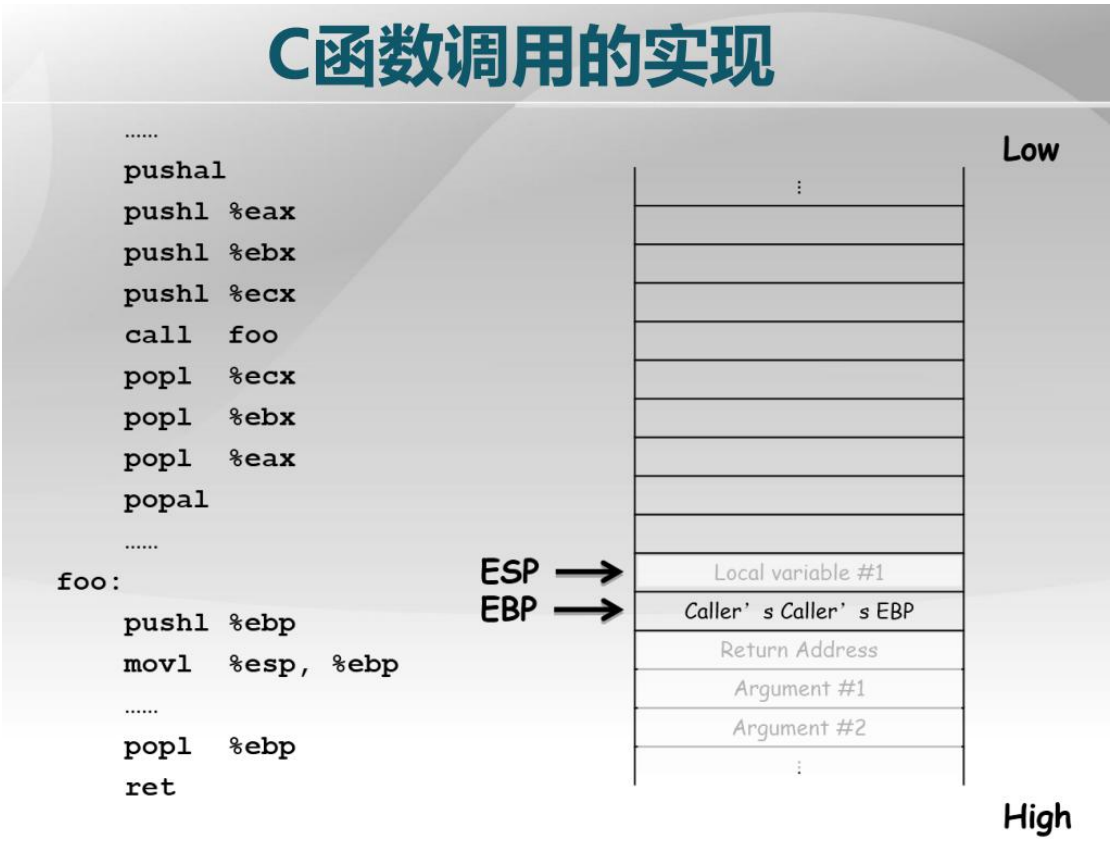
当 uCore 操作系统得到控制权后，它还要进一步执行，它需要了解函数调用关系(便于系统崩掉之后分析错误在什么地方)

GCC 如何产生代码，使得它可以把函数的参数压入到栈里面，具体被调函数能够获取这些参数去执行，最后还能正确返回。

1.理解 C 函数调用在汇编级是如何实现的

2.理解如何在汇编级代码中调用 C 函数

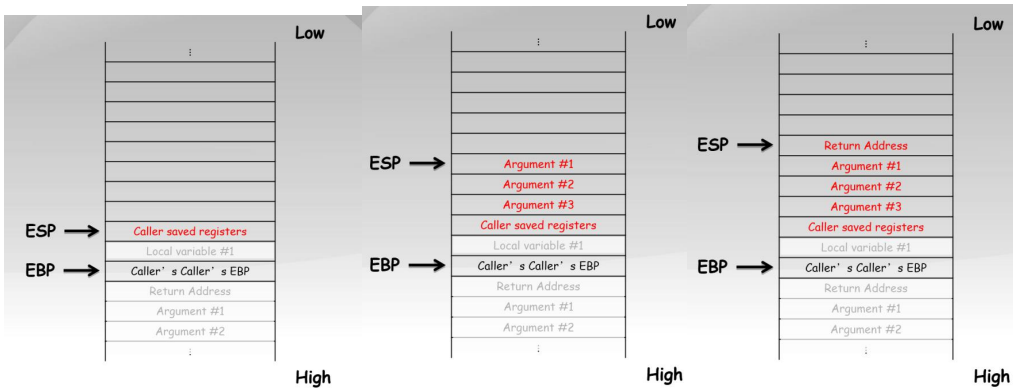
3.理解基于 EBP 寄存器的函数调用栈



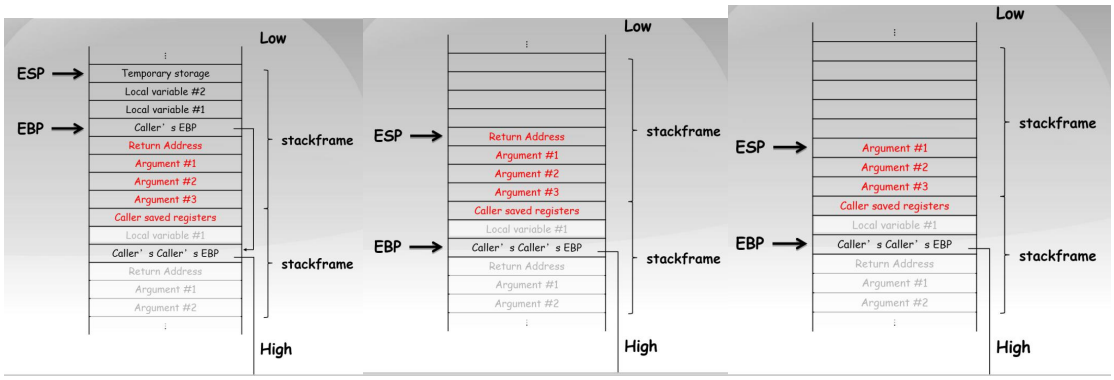
我们从机器码的角度进行分析。上图右侧所示是一个栈，在这个图上，标注了下面是高地址，上面是低地址。左侧的汇编代码是压栈->调用函数 foo(foo 可以指一个地址)->出栈的过程.

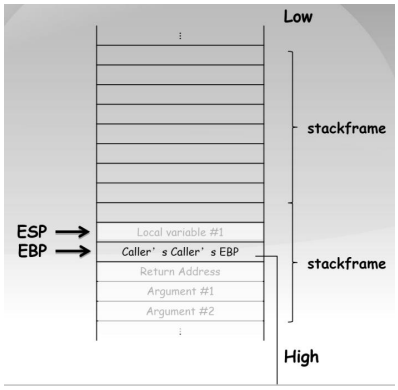
现对这段代码详细分析：pushal　此时 EBP 指向 caller's caller's EBP，当执行 pushal 时，实际上它把 caller，即调用 foo 这个函数的函数，就是 caller，它会把相应的寄存器压栈(即如下图左一，caller saved registers)

接下来，三个 push，会把相应的参数压栈，即将函数需要的参数压栈了(下图左二)
再接下来，如下图左三所示，调用函数 foo,在执行这个函数时，会将它的返回地址压栈(？？？返回地址从图三 EBP 下面那条语句中取出)



下图左一：压栈后跳转到 foo 函数去执行：注意在进行 push1 %ebp 时，EBP 的内容发生了变化，同时 mov1 %esp,%ebp，将 ESP 的内容付给了 EBP，于是 EBP 的位置发生了跳转，虽然 EBP 发生了跳转，但隐含链接关系:当前 EBP 中内容指向 Caller's Caller's EBP 所在地址，这就形成了所谓的调用栈的链。后面的 ESP 的操作是省略号部分代码的操作，不述。
左二：foo 执行到 pop1 %ebp,使得 EBP 回去了，在执行 ret 即 return 时，因为此时 ESP 已经指向 return address，这时会回到原位置到图三
在图三基础上执行三步 pop，便到图四。





注意：寄存器中存放地址，ESP 是栈顶寄存器，存放栈顶指针，%esp 是取 ESP 寄存器中的值，也就是一个地址，ESP 和 EBP 这些寄存器和栈那片地址空间本身并没有什么直接关系。push 才会改变栈中的值。

第三节 GCC 内联汇编

GCC 内联汇编是说怎么在 C 语言的开发环境中来使用汇编代码，使得C 和汇编混在一起使用，称之为内联汇编，这是 GCC 很有特点的地方。

什么是内联汇编（ Inline assembly ）?

这是 GCC 对 C 语言的扩张

可直接在 C 语句中插入汇编指令

有何用处?

调用 C 语言不支持的指令

用汇编在 C 语言中手动优化

如何工作?

用给定的模板和约束来生成汇编指令

在 C 函数内形成汇编源码

例一：

汇编(将值赋给 eax 寄存器)：

Assembly (*.S):

movl $0xffff, %eax

内联汇编(加了 asm 作为内联汇编的关键字，一个百分号变成了两个)：

Inline assembly (*.c):

asm (“movl $0xffff, %%eax\n”)

内联汇编的语法及完整格式：

```
asm ( assembler template       //刚才的字符串，即汇编代码，下面三个是约束表示
: output operands (optional)   //输出操作数
: input operands (optional)    //输入操作数
: clobbers (optional)          //暂时忽略
);
```

约束例：希望把某一个变量用某一个寄存器来表示，便可以在其中加一定的约束来完成这样的功能。

例子 2

```
Inline assembly (*.c):   //内联汇编
uint32_t cr0;            //一个 CR0 的内存变量，希望把 CR0 的某一个 bit 置位
asm volatile ("movl %%cr0, %0\n" :"=r"(cr0)); //首先把 CR0 的内容读到%0 寄存器中，这个寄存器中的内容赋给 cr0 这个变量，=r 是这个含义，unit32_t cr0 作为输出，其值来自于%0
cr0 |= 0x80000000;    //取或操作，把某位置成 1
asm volatile ("movl %0, %%cr0\n" ::"r"(cr0)); //某位置成 1 之后，再把内存变量里的内容写回到 CR0 寄存器中去,unit32_t cr0 作为输入，输入到%0 中

Generated asssembly code (*.s): //上面这段内联汇编代码生成的汇编代码
movl %cr0, %ebx   //CR0 寄存器的值赋给 ebx 寄存器
movl %ebx, 12(%esp) //ebx 的值赋给一个局部变量
```

orl $-2147483648, 12(%esp) //或操作，完成对某个 bit 的置 1
movl 12(%esp), %eax //将局部变量的值赋给 eax 寄存器
movl %eax, %cr0 //将 eax 赋给 cr0

上面内联汇编代码一些点的说明：

•volatile

No reordering; No elimination 不要调整顺序以及做进一步的优化

•%0

The first constraint following 表示第一个用到的寄存器

• r

A constraint; GCC is free to use any register 代表任意寄存器，GCC 有使用任何寄存器的自由

更复杂的一个例子：

例子 3：

```
long __res, arg1 = 2, arg2 = 22, arg3 = 222, arg4 = 233; //一系列的局部变量
__asm__ volatile ("int $0x80"   //执行 int 80 指令，产生软中断
: "=a" (__res)
: "0" (11),"b" (arg1),"c" (arg2),"d" (arg3),"S" (arg4)); //将 arg1 赋给 bx,arg2 赋给 cx 等等
```

以上等同于下面的汇编：会将这些局部变量赋给相应的寄存器，调用 int 80 这样一个指令，产生一个软中断，最后这个软中断的返回值被赋给_res(ebp)

```
……
movl $11, %eax
movl -28(%ebp), %ebx
movl -24(%ebp), %ecx
movl -20(%ebp), %edx
movl -16(%ebp), %esi
int $0x80
movl %eax, -12(%ebp)
```

以上的 a,b 等都代表寄存器：

```
a = %eax
b = %ebx
c = %ecx
d = %edx
S = %esi
D = %edi
0 = same as the first
```

第四节 x86 中断处理过程

1.了解 x86 中的中断源(谁产生了中断)
2.了解 CPU 与操作系统如何处理中断(软件和硬件如何结合在一起处理中断)
3.(怎样？)能够对中断向量表（中断描述符表，简称 IDT）进行初始化

X86 将中断和异常作为特定的两种不同类型来分别处理，但它们的实现机制却是统一的。

1.中断 Interrupts

外部中断(硬中断) External (hardware generated) interrupts：串口、硬盘、网卡、时钟、...

软件产生的中断 Software generated interrupts：The INT n 指令，通常用于系统调用(软中断，也叫内部中断)

2.异常 Exceptions(程序在执行过程中做了不该做的事情)

程序错误

软件产生的异常 Software generated exceptionsINTO, INT 3 and BOUND

机器检查出的异常 S

中断产生之后，操作系统怎么办？

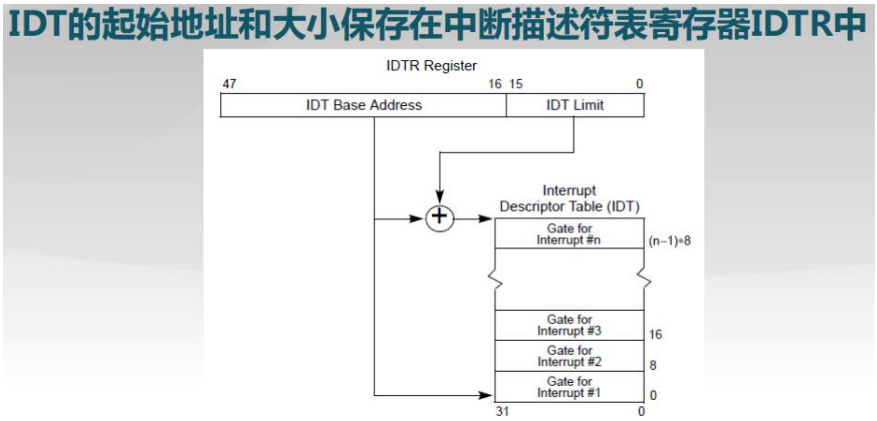
一般来说，中断是外设产生，操作系统应理解外设需要什么，并产生相应反馈。例：网卡产生了中断(网卡得到了一个数据包进而产生了中断),那么操作系统应对这个数据包做进一步的处理。来发给相应的应用程序(即操作系统要做相应的响应)

异常产生之后，操作系统要根据这个异常的严重程度，决定杀死(kill)掉这个进程，或者这个异常是应用程序产生的软中断，那么操作系统就可以去完成这个服务。

每个中断都有一个对应的中断号，中断号唯一标识了这个中断的特征，对于每一个中断号，都有相应的中断处理的一个例程来完成对应操作。

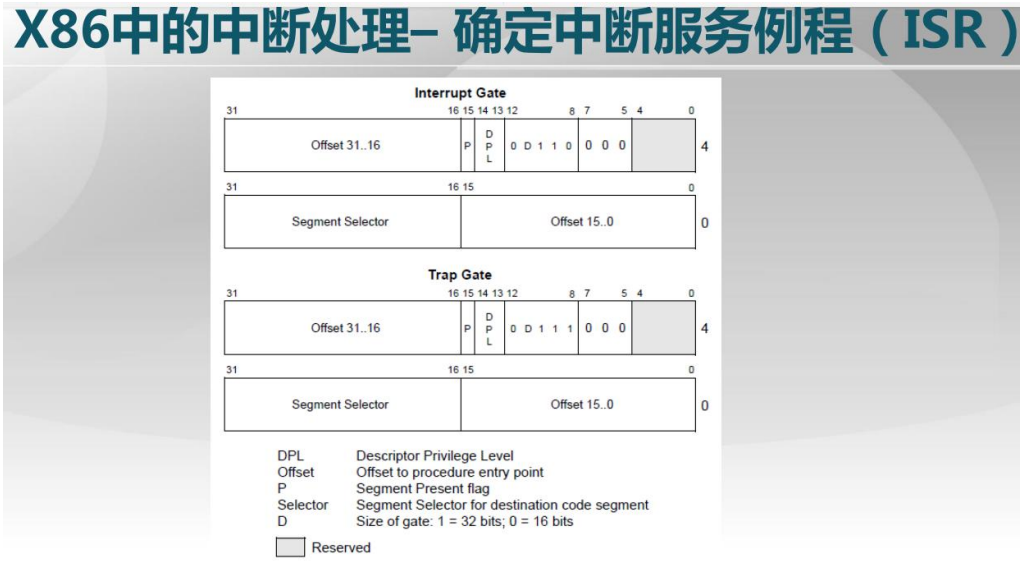
每个中断或异常与一个中断服务例程（ Interrupt Service Routine ，简称 ISR）关联，其关联关系存储在中断描述符表 （ Interrupt Descriptor Table，简称 IDT）中。

IDT 的起始地址和大小保存在中断描述符表寄存器 IDTR 中。



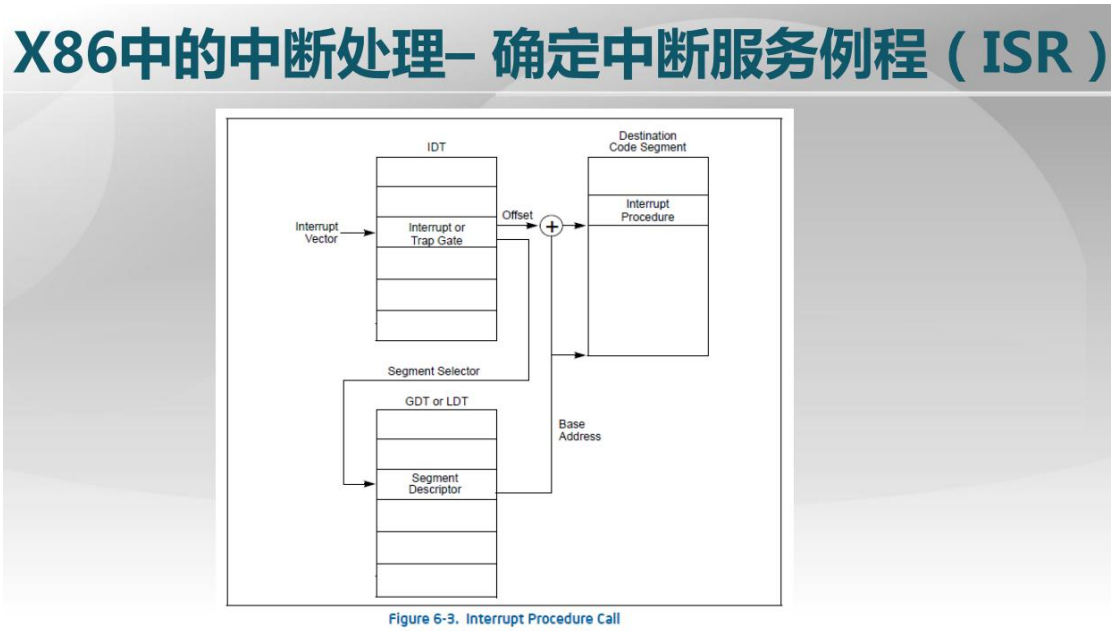
摘自"IA-32 Intel体系结构软件开发者手册"

IDT 实际上也是一个大数组，IDT 中的每一项我们称之为中断门或者陷阱门(trap 就是我们说的软中断)，中断门或者陷阱门对应相应的中断号，根据中断号可找到相应的中断门或者说陷阱门。由中断门或陷阱门，可进一步获取中断门陷阱门相关的段选择址(见第一节段机制的讲解)，从而得到中断服务例程的地址。



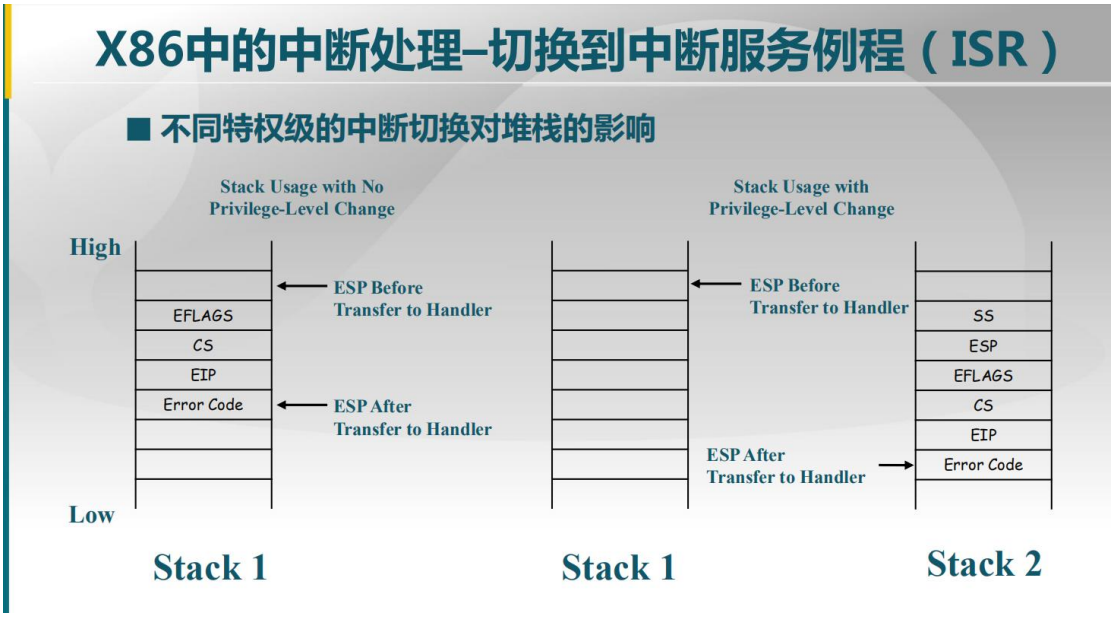
摘自"IA-32 Intel体系结构软件开发者手册"

对于 IDT 表中的每一项，即我们称之为中断门和陷阱门的，它们有一定格式，最主要的有：段描述符(Segment Selector)以及 offset(偏移),有了这两个信息，中断服务例程的地址也就知道了。



摘自"IA-32 Intel体系结构软件开发者手册"

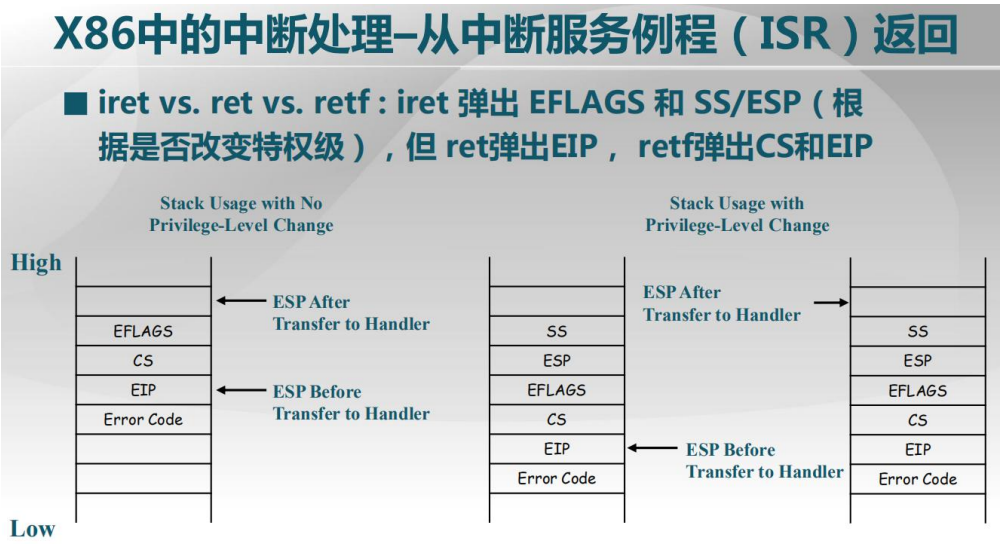
上图展示了，产生一个中断后，根据这个中断我们可以知道它的中断号，CPU 会根据这个中断号来查 IDT，判断属于 IDT 表中的哪一项，找到相应的中断门或者陷阱门，然后从中取出段选择址，以这个选择址作为 index(索引)进一步查找 GDT(全局描述符表)，找到段描述符，段描述符中有一个 Base Address(基地址)，Base Address 加上在 IDT 中存的 offset,就形成相应的线性地址，从而指向 ISR，即中断服务例程。所以一旦产生某个中断，CPU 可以自动在硬件层面访问这两个表(IDT、GDT)，注意，这两个表是 uCore 建立好的，建立好之后，CPU 就可以基于这两个表查到相应的中断需要对应的中断处理例程，这个例程是操作系统实现的 。



当产生中断之后，中断会打断当前正在执行的程序，然后去执行中断服务例程，执行完毕之后再返回到当前被打断的程序，继续让这个程序去执行。打断后要保存，返回要恢复。而不同特权级处理方式是不同的，特权级在段描述符中可以看到。如 CS 的低两位是 0，代表运行在内核态，低两位是 3，代表运行在用户态。在内核态产生的中断依然在内核态，但在用户态产生的中断会跳到内核态中，前一种特权级没变，后一种特权级变了。这两种涉及的中断的保存与恢复是不一样的。

上图左侧是产生中断后，在同一特权级，即内核态里产生的中断依然在内核态。分析：首先它的 Stack(栈)没有变，还是用的同一个栈，只是在这个栈上压了一些寄存器的内容，即被打断那一刻寄存器的内容：Error code,代表特意的严重的异常，不是每一个中断或异常都会产生 Error code.EIP 和 CS，是当前被打断的那个地址，或者是当前被打断的下一条地址.EFLAGS,当前被打断时的标志性内容。以上是产生中断时硬件会压栈压进去的内容。

上图右侧是发生中断时处于不同特权级，意味着产生中断那一刻，我们的应用程序在用户态执行。可以看到，从用户态到内核态，用的是不同的栈。因此当产生中断，特权级变化了时，除了压刚才说的那些内容，还有很重要的两个信息：ESP 和 SS，这两个内容是当时产生中断时，在用户态里栈的地址。



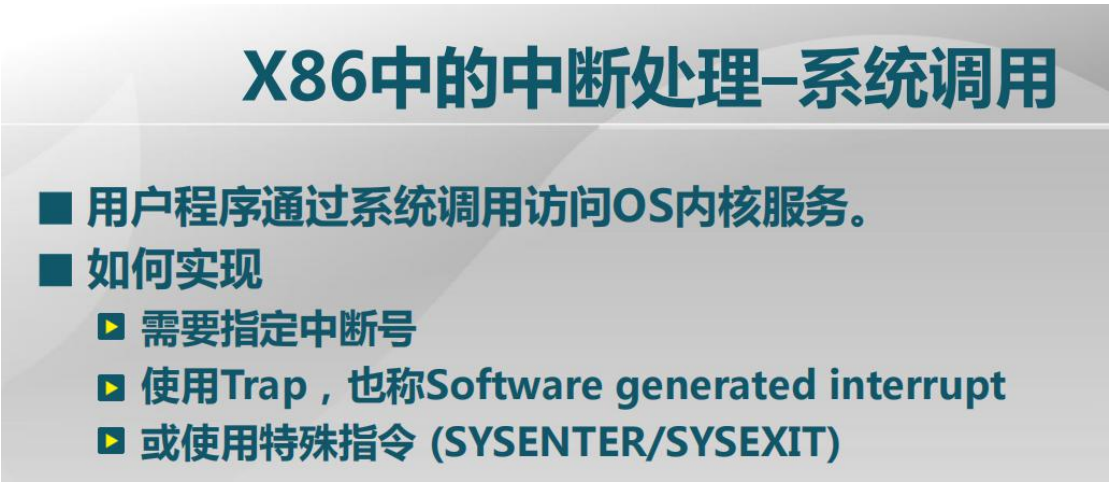
当 X86 完成中断服务例程的处理之后，需要返回被打断程序继续执行，这里对于中断服务例程来说，会通过一个 iret 指令来完成这个返回。但对于通常的程序，是通过 ret 和 retf 完成函数的返回。

iret:没有改变特权级的中断返回：根据 CS 和 EIP 跳到当前被打断的地方继续执行，同时要恢复 EFLAGS 的值　特权级变化的中断返回：EIP CS EFLAGS ESP SS 都要恢复，从而确保被打断的用户态的程序能够正常的继续执行

ret:只是弹出了 EIP，即跳到当时的调用指令的下一条指令去执行

retf:除了弹出 EIP，还会弹出 CS，恢复 CS，实现远程跳转

以上只是硬件完成的功能，软件方面来说：如果中断服务例程对其它寄存器进行修改，在修改之前，中断服务例程还需要把寄存器先保存起来，在 iret 返回时，需要把寄存器恢复回来，从而确保当跳回到这个被中断的应用程序时可以正常执行



通过中断处理来实现系统调用：系统调用可以理解为一种特殊的中断，称之为 trap(陷入或者叫软中断)

用户程序通过系统调用访问 OS 内核服务(lab5 内容)。从具体实现上来说，系统调用机制的建立和中断机制的建立是很接近的，基本没什么区别。

■如何实现系统调用？

需要指定中断号(如何指定中断？如何完成从用户态到内核态的切换？如何从内核态回到用户态？)

答：使用 Trap，也称 Software generated interrupt 或使用特殊指令 (SYSENTER/SYSEXIT)。在 uCore 中使用的还是传统的嵌入的方式，比如 int 80,即通过软中断的方式来完成系统调用，为了完成系统调用，需要在建立 IDT(中断描述符表)时，要特殊考虑，因为这与其它中断处理不太一样，因为这里很明确的指出了是从用户态执行 int 80 而切换到内核态，有一个从低优先级到高优先级的转变，这个机制需要我们在 IDT 表设置好相应的权限才能完成这种转变。

第五节 练习一

操作系统是一个软件，也需要通过某种机制加载并运行它。在这里我们将通过另外一个更加简单的软件-bootloader 来完成这些工作。为此，我们需要完成一个能够切换到 x86 的保护模式并显示字符的 bootloader，为启动操作系统 ucore 做准备。lab1 提供了一个非常小的 bootloader 和 ucore OS，整个 bootloader 执行代码小于 512 个字节，这样才能放到硬盘的主引导扇区中。通过分析和实现这个 bootloader 和 ucore OS，读者可以了解到：

计算机原理

CPU 的编址与寻址: 基于分段机制的内存管理

CPU 的中断机制

外设：串口/并口/CGA(我们的电脑显示器)，时钟(产生时钟中断)，硬盘(读取)

Bootloader 软件

编译运行 bootloader 的过程

调试 bootloader 的方法

PC 启动 bootloader 的过程

ELF 执行文件的格式和加载

外设访问：读硬盘，在 CGA 上显示字符串

让 CPU 进入保护模式

Bootloader 会把 ucoreOs 加载进来，然后把控制权交给 ucoreos 去运行

ucore OS 软件

编译运行 ucore OS 的过程

ucore OS 的启动过程

调试 ucore OS 的方法

函数调用关系：在汇编级了解函数调用栈的结构和处理过程

中断管理：与软件相关的中断处理(1.外设发出请求之后，CPU 可以通过中断机制得到相应 2.当产生异常或发出系统调用时，通过中断管理机制能感知到这种现象存在并进行相应的处理)

外设管理：时钟(让外设定期的产生时钟中断)

实验内容：

lab1 中包含一个 bootloader 和一个 OS。这个 bootloader 可以切换到 X86 保护模式，能够读磁盘并加载 ELF 执行文件格式，并显示字符。而这 lab1 中的 OS 只是一个可以处理时钟中断和显示字符的幼儿园级别 OS。

博客：https://blog.csdn.net/qq_31481187/article/details/63251420

练习 1：理解通过 make 生成执行文件的过程。

列出本实验各练习中对应的 OS 原理的知识点，并说明本实验中的实现部分如何对应和体现了原理中的基本概念和关键知识点。

在此练习中，大家需要通过静态分析代码来了解：

1.操作系统镜像文件 ucore.img 是如何一步一步生成的？(需要比较详细地解释 Makefile 中每一条相关命令和命令参数的含义，以及说明命令导致的结果)

答案见补充材料的 make "V="部分，现在先不用管 Makefile 文件，比较复杂。

2.一个被系统认为是符合规范的硬盘主引导扇区的特征是什么？

BIOS 在引导主引导扇区时怎么知道主引导扇区是符合规范的？这个规范是什么？

这需要阅读 tools/sign.c 文件，该文件中描述了这些特征，见博客。

补充材料：

如何调试 Makefile

当执行 make 时，一般只会显示输出，不会显示 make 到底执行了哪些命令。

如想了解 make 执行了哪些命令，可以执行：

$ make "V=" (设置一个标记，使得 Make 的执行过程能够展现出来)

执行后可以看到，它调用了 GCC，GCC 把 C 的源代码编译成了.O 即目标文件：

gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootasm.S -o obj/boot/bootasm.o

+ cc boot/bootmain.c

gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootmain.c -o obj/boot/bootmain.o

+ cc tools/sign.c

然后通过 ld 把这些目标文件转换成执行程序(bootblock.out)：

ld -m elf_i386 -nostdlib -N -e start -Ttext 0x7C00 obj/boot/bootasm.o obj/boot/bootmain.o -o obj/bootblock.o

'obj/bootblock.out' size: 488 bytes

build 512 bytes boot sector: 'bin/bootblock' success!

dd 可以把 bootloader 放到一个虚拟的硬盘里面去，这里生成一个虚拟硬盘叫 uCore.img,qemu 硬件模拟器会基于虚拟硬盘中的数据来执行相应的代码:

dd if=/dev/zero of=bin/ucore.img count=10000

后面还可以看到这里生成了两个软件，第一个是 bootloader,第二个是 kernel,kernel 实际上是 uCore 的组成部分：

dd if=bin/bootblock of=bin/ucore.img conv=notrunc

dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc

要获取更多有关 make 的信息，可上网查询，并请执行

$ man make

练习二

执行：make lab1-mon，在 Makefile 文件中可以找到该命令的定义，首先它将 qemu 的执行指令记录下来放在一 q.log 文件中,其次和 GDB 结合来调试正在执行的 bootloader，调试指令可以看到存放在 tools/lab1init 中：

file bin/kernel //加载 kernel(加载符号信息，实际上是 uCore 的信息，现在还未用到)

target remote :1234 //与 qemu 通过 TRP 进行连接

set architecture i8086 //刚开始 BIOS 进入的是 8086 的 16 位实模式方式

b *0x7c00 //BIOS 将控制权交给 bootloader,bootloader 的第一条指令在 0x7c00 处执行，在此处设置断点

Continue //继续运行

x /2i $pc //x:显示 /2i:两条指令 $pc:指令地址 显示两条指令及地址

可以看到在 gdb 窗口中，在 0x7c00 处中断，可以用指令：x /10i $pc 显示中断处以下的更多指令(10 条)，打开 boot 目录下的 bootasm.S 文件，可以发现显示的 10 条指令和 bootasm.S 文件中的从 16 行开始的指令是一样的，也就是说我们目前的断点在 bootloader 的起始位置

然后输入指令：continue,在 qemu 窗口，可见程序运行起来了，uCore 被加载了进来。

练习三

Bootloader 可以把 80386 的保护模式开启，使得软件进入 32 位的寻址空间(寻址方

式发生了改变)，为了完成这一目标，需要进行三步：

开启 A20　　初始化 GDT 表　　使能和进入保护模式

以上三步的实现过程见 bootasm.S 文件的 16-56 行

tools/sign.c 是一个生成主引导扇区的辅助工具，通过它可以生成一个合格的 Bootloader 主引导扇区

练习四 五

ELF 文件：我们的 uCore OS 是以 ELF 格式存在硬盘上的，即 bin/kernel 这个文件，它完成了初始化中断、产生时钟中断、把信息显示在屏幕上等工作。

bootloader 如何读取硬盘扇区的？(bootloader 需要能读取硬盘)

bootloader 是如何加载 ELF 格式的 OS？

堆栈跟踪函数

练习六

中断向量表：了解中断向量表的一项包含哪些中断描述符，中断描述符由什么信息组成，和中断相关的在 kern/trap/trap.c 文件以及 kern/init/init.c 文件中

Kern/trap/vectors.S 定义了中断号及其中断服务例程的入口地址。

vectors.S 中最后总会跳到_alltraps，alltraps 在 kern/trap/trapentry.S 中，这里保存了一系列寄存器。