Ucore lab1

叶明宇

这个Task主要是为了熟悉GDB以及熟悉操作系统的启动过程，下面是调试BIOS的一些过程。

首先修改gdb init为:

set architecture i8086

target remote :1234define hook-stop

x/i $pcend

然后输入

make debug

通过输入

x/i $csx/i $eip

我们可以获取当前 $cs 和 $eip 的值。其中

$cs = 0xf000

$eip = 0xfff0

在实模式下，这个地址就是

$cs << 4 | $eip = 0xffff0

我们也可以看看这个地址的指令是什么

x/2i 0xffff0

得到的结果是

0xffff0: ljmp $0xf000,$0xe05b

也就是说，BIOS开始的地址应该是

$cs << 4 | 0xe05b = 0xfe05b

此时, 我们设置一个断点到0x7c00:

b \*0x7c00 /\* 注意，对于绝对地址来说，需要添加\*将其作为地址 \*/

然后当程序运行起来后, 最后会停止在 0x7c00 这个地址。这里存放的便是bootloader了。

## Task3

这个Taks是这5个Taks中最重要的一个。通过这个Task我们可以了解：如何开启A20；CPU是如何从实模式转换到保护模式；如何初始化和使用GDT表。

### 如何开启/关闭 A20

#### 实模式下内存的访问

在开启A20前，我们先来说说i8086时CPU是如何访问内存空间的。

在i8086时代，CPU的数据总线是16bit，地址总线是20bit，寄存器是16bit，因此CPU只能访问1MB以内的空间。因为数据总线和寄存器只有16bit，如果需要获取20bit的数据, 我们需要做一些额外的操作，比如移位。实际上，CPU是通过对segment(每个segment大小恒定为64K) 进行移位后和offset一起组成了一个20bit的地址，这个地址就是实模式下访问内存的地址：

address = segment << 4 | offset

理论上，20bit的地址可以访问1MB的内存空间(0x00000 - (2^20 - 1 = 0xFFFFF))。但在实模式下, 这20bit的地址理论上能访问从0x00000 - (0xFFFF0 + 0xFFFF = 0x10FFEF)的内存空间。也就是说，理论上我们可以访问超过1MB的内存空间，但越过0xFFFFF后，地址又会回到0x00000。

上面这个特征在i8086中是没有任何问题的(因为它最多只能访问1MB的内存空间)，但到了i80286/i80386后，CPU有了更宽的地址总线，数据总线和寄存器后，这就会出现一个问题： 在实模式下, 我们可以访问超过1MB的空间，但我们只希望访问1MB以内的内存空间。为了解决这个问题， CPU中添加了一个可控制A20地址线的模块，通过这个模块，我们在实模式下将第20bit的地址线限制为0，这样CPU就不能访问超过1MB的空间了。进入保护模式后，我们再通过这个模块解除对A20地址线的限制，这样我们就能访问超过1MB的内存空间了。

#### A20开启/关闭的过程

现在使用的CPU都是通过键盘控制器8042来控制A20地址线。默认情况下，A20地址线是关闭的(第20bit的地址线限制为0)，因此在进入保护模式(需要访问超过1MB的内存空间)前，我们需要开启A20地址线(第20bit的地址线可为0或者1)。A20的开启过程请参考bootasm.S文件。

### CPU是如何从实模式转换到保护模式

这个特别简单，我们需要在开启A20地址线后，将$CR0(control register 0)的PE(bit0)置为1就行了。具体代码请参考bootasm.S文件。

### 如何初始化和使用GDT表

#### GDT详解

在使用GDT前，我们需要先来了解什么是GDT。GDT全称是Global Descriptor Table，也就是全局描述符表。在保护模式下，我们通过设置GDT将内存空间被分割为了一个又一个的segment(这些segment是可以重叠的)，这样我们就能实现不同的程序访问不同的内存空间。  
这和实模式下的寻址方式是不同的, 在实模式下我们只能使用

address = segment << 4 | offset

的方式进行寻址(虽然也是segment + offset的，但在实模式下我们并不会真正的进行分段)。在这种情况下，任何程序都能访问整个1MB的空间。而在保护模式下，通过分段的方式，程序并不能访问整个内存空间。下面引用一段ucore实验报告书上的说明：

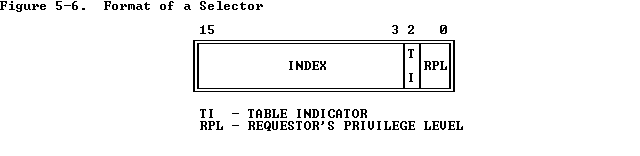
【补充】保护模式下，有两个段表：GDT（Global Descriptor Table）和LDT（Local Descriptor Table），每一张段表可以包含8192 (2^13)个描述符[1]，因而最多可以同时存在2 \* 2^13 = 2^14个段。虽然保护模式下可以有这么多段，逻辑地址空间看起来很大，但实际上段并不能扩展物理地址空间，很大程度上各个段的地址空间是相互重叠的。目前所谓的64TB（2^(14+32)=2^46）逻辑地址空间是一个理论值，没有实际意义。在32位保护模式下，真正的物理空间仍然只有2^32字节那么大。注：在ucore lab中只用到了GDT，没有用LDT。

Reference: [1] 3.5.1 Segment Descriptor Tables, Intel® 64 and IA-32 Architectures Software Developer’s Manual

除了GDT, 我们还需要了解另外几个名词：段描述符(segment descriptor)和段选择子(segment selector)。段描述符就是GDT中的元素，段选择子就是访问GDT的索引。

##### 段选择子

在实模式下, 逻辑地址由段选择子和段选择子偏移量组成. 其中, 段选择子16bit, 段选择子偏移量是32bit. 下面是段选择子的示意图:



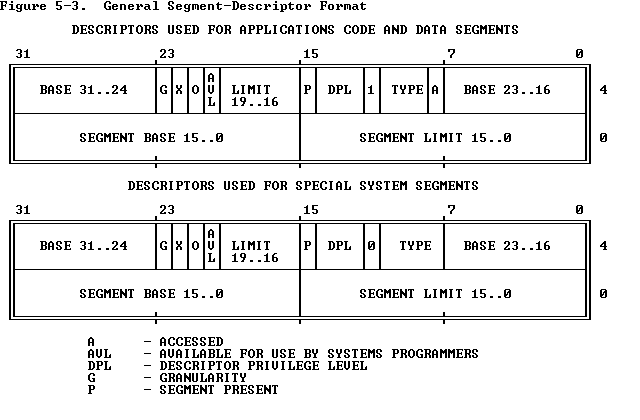
在段选择子中，其中的INDEX[15:3]是GDT的索引

TI[2:2]用于选择表格的类型，1是LDT，0是GDT。

RPL[1:0]用于选择请求者的特权级，00最高，11最低。

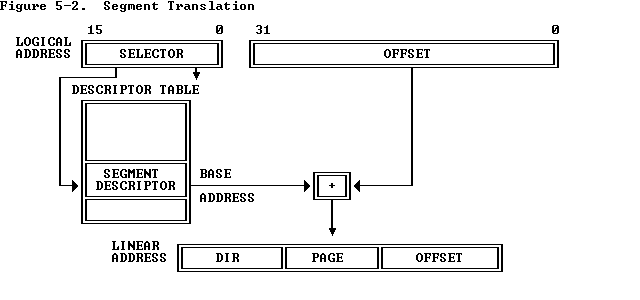
##### 段描述符

段描述符的形式比较复杂(为了兼容各种不同版本的CPU)，这里我只给一个示意图，具体的内容请查找手册。这里用到的最重要的是segment base和segment limit：



##### GDT的访问

有了上面这些知识，我们可以来看看到底应该怎样通过GDT来获取需要访问的地址了。我们通过这个示意图来讲解：



我们根据CPU给的逻辑地址分离出段选择子。

利用这个段选择子选择一个段描述符。

将段描述符里的Base Address和段选择子的偏移量相加而得到线性地址。这个地址就是我们需要的地址。

#### GDT的初始化和使用

因为在保护模式下我们需要使用分段的内存空间，因此在进入保护模式前，我们就需要初始化GDT。 下面就通过一些代码来说明如何初始化和使用GDT。

下面是GDT初始化的代码:

#define SEG\_NULLASM

.word 0, 0;

.byte 0, 0, 0, 0

#define SEG\_ASM(type,base,lim)

.word (((lim) >> 12) & 0xffff), ((base) & 0xffff);

.byte (((base) >> 16) & 0xff), (0x90 | (type)),

(0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)

gdt:

/\* 有一个特殊的选择子称为空(Null)选择子，它的Index=0，TI=0，而RP

L字段可以为任意值。空选择子有特定的用途，当用空选择子进行存储访

问时会引起异常。空选择子是特别定义的，它不对应于全局描述符表GDT

中的第0个描述符，因此处理器中的第0个描述符总不被处理器访问，一

般把它置成全0。\*

SEG\_NULLASM # null seg

/\* 在Lab1中, code segment和data segment都可以访问整个内存空间 \*

SEG\_ASM(STA\_X|STA\_R, 0x0, 0xffffffff) # code seg for bootloader and kernel

SEG\_ASM(STA\_W, 0x0, 0xffffffff) # data seg for bootloader and kernel

gdtdesc:

/\* lgdt 要先载入GDT的大小, 然后才是gdt的地址 \*/

.word 0x17 # sizeof(gdt) - 1

.long gdt # address gdt

理论上GDT可以存在内存中任何位置，但这里我们是在实模式下初始化GDT的，因此GDT应该是存在最低的这1MB内存空间中。CPU通过lgdt指令读入GDT的地址，之后我们就可以使用GDT了。

.set PROT\_MODE\_CSEG, 0x8

.set PROT\_MODE\_DSEG, 0x10

/\* 载入GDT \*/

lgdt gdtdesc

/\* 从实模式切换到保护模式\*/

movl %cr0, %eax

orl $CR0\_PE\_ON, %eax

movl %eax, %cr0

# ljmp <imm1>, <imm2># %cs ← imm1# %ip ← imm2/\* 将%cs(code segment)的值设置为0x8 \*/

ljmp $PROT\_MODE\_CSEG, $protcseg

...

protcseg:

# Set up the protected-mode data segment registers

/\* 设置data segment 的值 \*/

movw $PROT\_MODE\_DSEG, %ax # Our data segment selector

movw %ax, %ds # -> DS: Data Segment

movw %ax, %es # -> ES: Extra Segment

movw %ax, %fs # -> FS

movw %ax, %gs # -> GS

movw %ax, %ss # -> SS: Stack Segment

## Task4

通过这个Task，我们可以了解OS是如何加载ELF镜像文件的。这里我并没有仔细研究ELF文件格式以及如何使用。

## Task5

这个task是为了让我们了解函数的调用和堆栈的关系。对于函数调用的细节，我在之前的文章中已经写过了，具体请参见[C函数调用过程原理及函数栈帧分析](https://segmentfault.com/a/1190000007977460)。这里主要分析下代码，源代码在 kern/debug/kdebug.c文件中。

/\*

栈底方向 高位地址

...

...

参数3

参数2

参数1

返回地址

上一层[ebp] <-------- [esp/当前ebp]

局部变量 低位地址

\*/voidprint\_stackframe(void) {

uint32\_t cur\_ebp, cur\_eip;

uint32\_t args[4];

cur\_ebp = read\_ebp();

cur\_eip = read\_eip();

/\* 假设最多有20层的函数调用 \*/

for (int stack\_level = 0; stack\_level < STACKFRAME\_DEPTH + 1; stack\_level++) {

cprintf("ebp: 0x%08x eip: 0x%08x ", cur\_ebp, cur\_eip);

/\* 假设函数最多有4个参数 \*/

for (int arg\_num = 0; arg\_num < 4; arg\_num++)

args[arg\_num] = \*((uint32\_t \*)cur\_ebp + (2 + arg\_num));

cprintf("args:0x%08x 0x%08x 0x%08x 0x%08x\n", args[0], args[1], args[2], args[3]);

print\_debuginfo(cur\_eip);

/\* 获取上一层函数的返回地址和$ebp的值 \*/

cur\_eip = \*((uint32\_t \*)cur\_ebp + 1);

cur\_ebp = \*((uint32\_t \*)cur\_ebp);

}

}

## Task6

这个Task主要是为了让我们熟悉保护模式下的中断。在X86架构中，中断可以分为3种：

和CPU无关的，比如外设的请求等，这些属于Interrupt。

和CPU有关的，比如除0，page fault等，这些属于Exception。

系统调用，这些属于Trap

### 中断机制

当CPU收到中断(通过8259A完成)或者异常的事件时，它会暂停执行当前的程序或任务，通过一定的机制跳转到负责处理这个信号的相关处理例程中，在完成对这个事件的处理后再跳回到刚才被打断的程序或任务中.

### 中断向量和中断服务例程

在X86架构中, 系统最多支持256种不同的中断， 这些中断都有一个相应的中断向量与其对应. 每个中断向量又有一个对应的中断服务例程， 这个中断服务例程用于处理中断向量.

### IDT

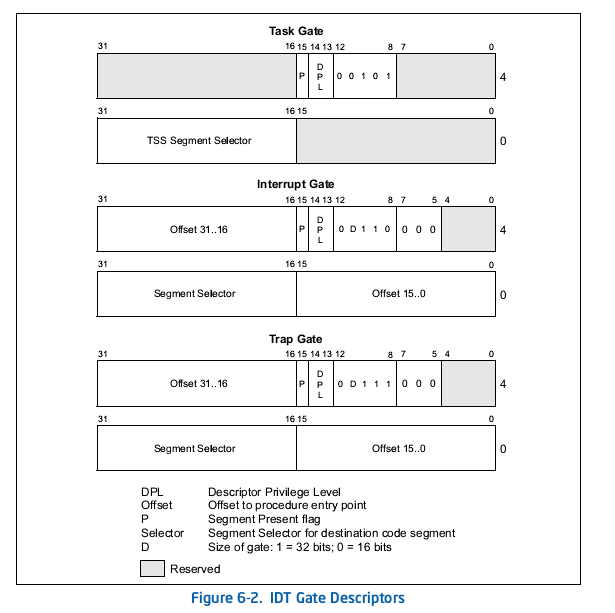
将中断向量和中断服务例程联系在一起的是IDT(Interrupt Descriptor Table)，输入一个中断向量，我们可以找到并运行该中断向量对应的中断服务例程。IDT和GDT类似，每个描述符都是8K，但IDT的第一项可以包含一个描述符。IDT中的中断描述符可以分为3种：

Task Gate

Interrupt Gate

Trap Gate

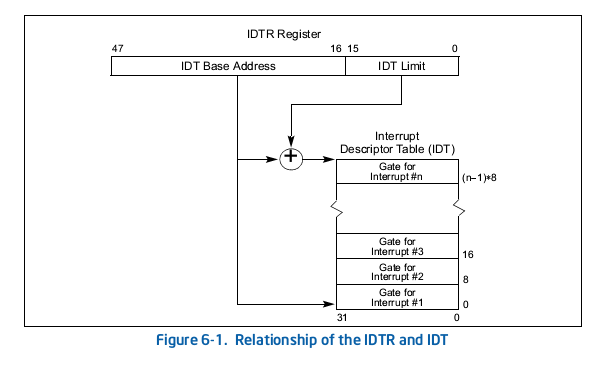
在这个Lab中我们使用了后两种中断描述符.



Interrupt Gate和Trap Gate差不多，但有些微小的区别，我直接引用老师的说明:

【补充】所谓“自动禁止”，指的是CPU跳转到interrupt gate里的地址时，在将EFLAGS保存到栈上之后，清除EFLAGS里的IF位，以避免重复触发中断。在中断处理例程里，操作系统可以将EFLAGS里的IF设上,从而允许嵌套中断。但是必须在此之前做好处理嵌套中断的必要准备，如保存必要的寄存器等。二在ucore中访问Trap Gate的目的是为了实现系统调用。用户进程在正常执行中是不能禁止中断的，而当它发出系统调用后，将通过Trap Gate完成了从用户态（ring 3）的用户进程进了核心态（ring 0）的OS kernel。如果在到达OS kernel后禁止EFLAGS里的IF位，第一没意义（因为不会出现嵌套系统调用的情况），第二还会导致某些中断得不到及时响应，所以调用Trap Gate时，CPU则不会去禁止中断。总之，interrupt gate和trap gate之间没有优先级之分，仅仅是CPU在处理中断时有不同的方法，供操作系统在实现时根据需要进行选择。

根据实际需求，我们建立相应的IDT，在建立好IDT后，我们就需要告诉CPU我们建立的IDT在哪里。要实现这个目的，我们需要使用一个专门的指令lidt将IDT的地址加载到IDTR寄存器中。这样 CPU就通过这个寄存器便可以访问IDT了。在IDTR寄存器中，我们需要存入IDT的起始地址和大小。下面是IDTR寄存器的示意图：



### 中断实例

我这里通过该Task的代码来说明如何建立IDT以及如何通过中断向量来访问相应的中断服务例程。

#### 建立中断向量表

在这个lab中，中断向量表是\_\_vectors，该表的每一项存储一个中断向量的地址。中断服务例程在\_\_alltraps中被调用。 \_\_alltraps除了调用中断服务例程外，还会做现场保护等工作。

# kern/trap/vectors.S.globl vector0vector0:

pushl $0

pushl $0

jmp \_\_alltraps

....globl vector255vector255:

pushl $0

pushl $255

jmp \_\_alltraps

# vector table.data.globl \_\_vectors\_\_vectors:

.long vector0

.long vector1

.long vector2

.long vector3

...

.long vector255

# kern/trap/trapentry.S.globl \_\_alltraps\_\_alltraps:

... # push %esp to pass a pointer to the trapframe as an argument to trap() # 我这里补充一下, 在call \_\_alltraps 之前, $esp指向最后压入的一个参数, 也就是interrupt number(比如pushl $255). 所以说这里 pushl %esp 就是把 $255 在stack中的地址压入stack作为 trap() 的参数

pushl %esp

# call trap(tf), where tf=%esp

call trap

#### 建立IDT

在这个Lab中，前32个中断向量和T\_SYSCALL使用的是Trap Gate；其余的中断向量都是使用Interrupt Gate。

voididt\_init(void) {

extern uintptr\_t \_\_vectors[];

for (int i = 0; i < 256; i++) {

if (i < IRQ\_OFFSET) {

SETGATE(idt[i], 1, GD\_KTEXT, \_\_vectors[i], DPL\_KERNEL);

} else if (i == T\_SYSCALL) {

SETGATE(idt[i], 1, GD\_KTEXT, \_\_vectors[i], DPL\_USER);

} else {

SETGATE(idt[i], 0, GD\_KTEXT, \_\_vectors[i], DPL\_KERNEL);

}

}

lidt(&idt\_pd);

}

#### 中断处理流程

下图是一个简化版的中断处理流程：

当系统接收到中断后, 会根据中断类型产生一个中断向量。

用这个中断向量作为索引在IDT中找到相应的中断描述符。

利用中断描述符中的Segment Selector在GDT中找到相应的Segment。

将3中找到的Segment和中断描述符中的Offset(也就是中断向量表中存储的中断向量的地址)相加得到中断服务例程的地址。

调用这个中断服务例程。