# 操作系统实验报告 lab1

姓名 朱凯 学号4042017046

* 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)**
  2. 利用make V= 查看执行了那些命令

### 生成ucore.img的代码如下

$(UCOREIMG): $(kernel) $(bootblock)

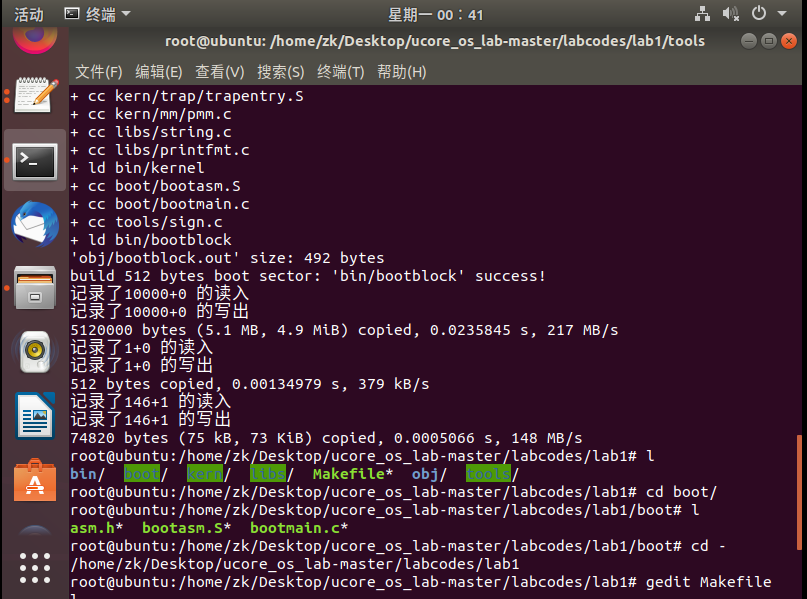
$(V)dd if=/dev/zero of=$@ count=10000

$(V)dd if=$(bootblock) of=$@ conv=notrunc

$(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc

$(call create\_target,ucore.img)

输出如下图



指令：   
dd：用指定大小的块拷贝一个文件，并在拷贝的同时进行指定的转换。   
if=文件名：输入文件名，缺省为标准输入。即指定源文件。< if=input file >   
of=文件名：输出文件名，缺省为标准输出。即指定目的文件。< of=output file >   
count=blocks：仅拷贝blocks个块，块大小等于ibs指定的字节数。   
conv=conversion：用指定的参数转换文件。   
conv=notrunc:不截短输出文件

由上描述可以看出，首先先创建一个大小为10000字节的块，然后再将bootblock，kernel拷贝过去。然而生成ucore.img需要先生成kernel和bootblock

#### 1.生成bootblock的相关代码如下

$(bootblock): $(call toobj,$(bootfiles)) | $(call totarget,sign)

@echo "========================$(call toobj,$(bootfiles))"

@echo + ld $@

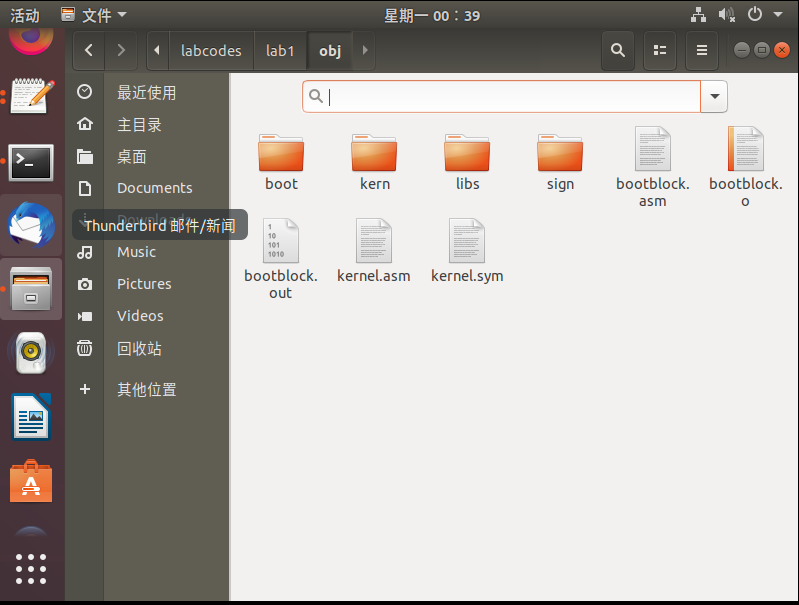
$(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ -o $(call toobj,bootblock)

@$(OBJDUMP) -S $(call objfile,bootblock) > $(call asmfile,bootblock)

@$(OBJCOPY) -S -O binary $(call objfile,bootblock) $(call outfile,bootblock)

@$(call totarget,sign) $(call outfile,bootblock) $(bootblock)

由上代码可得，到要生成bootblock，首先需要生成bootasm.o、bootmain.o、sign   
下图是在编译时生成的中间文件



1. 生成bootasm.o、bootmain.o、sign的相关代码为：

其中相关参数的含义为：   
ggdb 生成可供gdb使用的调试信息   
-m32生成适用于32位环境的代码   
-gstabs 生成stabs格式的调试信息   
-nostdinc 不使用标准库   
-fno-stack-protector 不生成用于检测缓冲区溢出的代码   
-0s 位减小代码长度进行优化

拷贝二进制代码bootblock.o到bootblock.out   
objcopy -S -O binary obj/bootblock.o obj/bootblock.out   
其中关键的参数为   
-S 移除所有符号和重定位信息   
-O 指定输出格式   
使用sign工具处理bootblock.out，生成bootblock   
bin/sign obj/bootblock.out bin/bootblock

kernel = $(call totarget,kernel)

$(kernel): tools/kernel.ld

$(kernel): $(KOBJS)

@echo + ld $@

$(V)$(LD) $(LDFLAGS) -T tools/kernel.ld -o $@ $(KOBJS)

@$(OBJDUMP) -S $@ > $(call asmfile,kernel)

@$(OBJDUMP) -t $@ | $(SED) '1,/SYMBOL TABLE/d; s/ .\* / /; /^$$/d' > $(call symfile,kernel)

$(call create\_target,kernel)

查看命令，生成kernel需要以下文件：

ld -m elf\_i386 -nostdlib -T tools/kernel.ld -o bin/kernel obj/kern/init/init.o obj/kern/libs/readline.o obj/kern/libs/stdio.o obj/kern/debug/kdebug.o obj/kern/debug/kmonitor.o obj/kern/debug/panic.o obj/kern/driver/clock.o obj/kern/driver/console.o obj/kern/driver/intr.o obj/kern/driver/picirq.o obj/kern/trap/trap.o obj/kern/trap/trapentry.o obj/kern/trap/vectors.o obj/kern/mm/pmm.o obj/libs/printfmt.o obj/libs/string.o

### 1.2 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？

查看sign.c源代码

buf[510] = 0x55;

buf[511] = 0xAA;

FILE \*ofp = fopen(argv[2], "wb+");size = fwrite(buf, 1, 512, ofp);

if (size != 512) {

fprintf(stderr, "write '%s' error, size is %d.\n", argv[2], size);

return -1;

}

从上述代码可以看出，要求硬盘主引导扇区的大小是512字节，还需要第510个字节是0x55,第511个字节为0xAA,也就是说扇区的最后两个字节内容是0x55AA



# 练习2

题目要求：   
从 CPU加电后执行的第一条指令开始，单步跟踪 BIOS的执行。   
在初始化位置 0x7c00 设置实地址断点,测试断点正常。   
从 0x7c00 开始跟踪代码运行,将单步跟踪反汇编得到的代码与 bootasm.S和 bootblock.asm进行比较。   
自己找一个 bootloader或内核中的代码位置，设置断点并进行测试

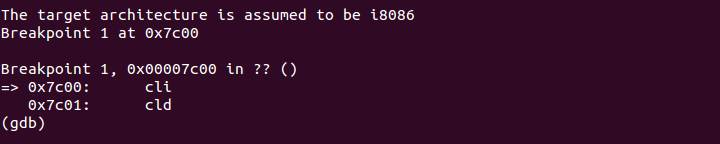
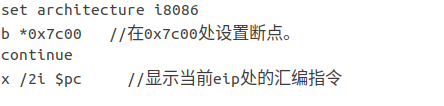
（1） 从 CPU 加电后执行的第一条指令开始,单步跟踪 BIOS 的执行

改写Makefile文件

在调用qemu时增加-d in\_asm -D q.log参数，便可以将运行的汇编指令保存在q.log

中。

（2） 在初始化位置0x7c00 设置实地址断点,测试断点正常。

在tools/gdbinit结尾加上

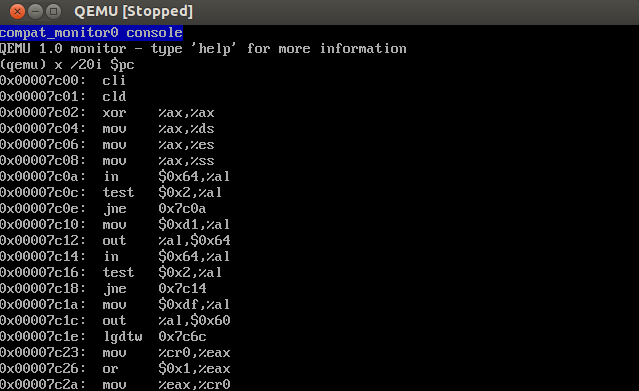
（3） 将执行的汇编代码与bootasm.S 和 bootblock.asm 进行比较，看看二者是否一致。

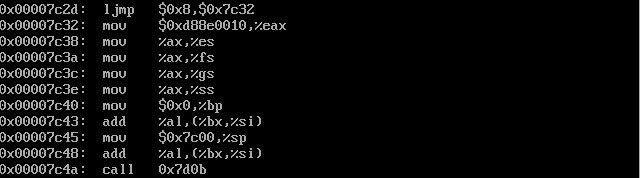
Notice：在q.log中进入BIOS之后的跳转地址与实际应跳转地址不相符，汇编代码也

与bootasm.S 和 bootblock.asm不相同。

这是由于在gdb之中调试的原因，可以直接输入make debug,在生成的qemu虚拟

机之中进行调试可以看到在虚拟机中运行的汇编代码，之后再与bootasm.S 和 bootblock.asm 进行比较。





# 练习3

题目：   
分析bootloader 进入保护模式的过程。   
BIOS 将通过读取硬盘主引导扇区到内存，并转跳到对应内存中的位置执行 bootloader。请分析bootloader是如何完成从实模式进入保护模式的

从bootasm.s查看代码(在这里分析bootblock.asm也可以，二者源码相同)，并分析过程

### 宏定义

.set PROT\_MODE\_CSEG, 0x8 #内核代码段选择子

.set PROT\_MODE\_DSEG, 0x10 #内核数据段选择子

.set CR0\_PE\_ON, 0x1 #保护模式使能标志

### 1.关闭中断，将各个段寄存器重置

修改控制方向标志寄存器DF=0，使得内存地址从低到高增加   
它先将各个寄存器置0

*#CPU刚启动为16位模式*

cli *# 关中断*

cld *# 清方向标志*

xorw %ax, %ax *# 置零*

movw %ax, %ds *# -> 数据段寄存器*

movw %ax, %es *# -> 附加段寄存器*

movw %ax, %ss *# -> 堆栈段寄存器*

### 2 .开启A20

开启A20地址线之后，用来表示内存地址的位数变多了。开启前20位，开启后是32位。如果不开启A20地址线内存寻址最大只能找到1M，对于1M以上的地址访问会变成对address mod 1M地址的访问。通过将键盘控制器上的A20线置于高电位，全部32条地址线可用，可以访问4G的内存空间。   
打开A20地址线 为了兼容早期的PC机，第20根地址线在实模式下不能使用所以超过1MB的地址，默认就会返回到地址0，重新从0循环计数，下面的代码打开A20地址线 。

seta20.1:

inb $0x64, %al *# 从0x64端口读入一个字节的数据到al中*

testb $0x2, %al *# test指令可以当作and指令，只不过它不会影响操作数*

jnz seta20.1*#如果上面的测试中发现al的第2位为0，就不执行该指令*

否则就循环检查

movb $0xd1, %al *# 将0xd1写入到al中*

outb %al,$0x64 *#将al中的数据写入到端口0x64中*

seta20.2:

inb $0x64, %al

testb $0x2, %al

jnz seta20.2

movb $0xdf, %al *# 通过0x60写入数据11011111 即将A20置1*

outb %al, $0x60

初始化GDT表：一个简单的GDT表和其描述符已经静态储存在引导区中，载入即可

lgdt gdtdesc *#将全局描述符表描述符加载到全局描述符表寄存器*

* 1

进入保护模式：通过将cr0寄存器PE位置1便开启了保护模式

cr0中的第0位为1表示处于保护模式

cr0中的第0位为0，表示处于实模式

把控制寄存器cr0加载到eax中

movl %cr0, %eax

orl $CR0\_PE\_ON, %eax

movl %eax, %cr0

8

通过长跳转更新cs的基地址

ljmp $PROT\_MODE\_CSEG, $protcseg

.code32

protcseg:

设置段寄存器，并建立堆栈

movw $PROT\_MODE\_DSEG, %ax

movw %ax, %ds

movw %ax, %es

movw %ax, %fs

movw %ax, %gs

movw %ax, %ss

movl $0x0, %ebp

movl $start, %esp

转到保护模式完成，进入boot主方法

call bootmain

# 

# 练习4

题目：   
分析bootloader加载ELF格式的OS的过程   
1. bootloader如何读取硬盘扇区的？   
2. bootloader是如何加载 ELF格式的 OS？

bootmain 代码

bootmain(void) {

readseg((uintptr\_t)ELFHDR, SECTSIZE \* 8, 0);

if (ELFHDR->e\_magic != ELF\_MAGIC) {

goto bad;

}

struct proghdr \*ph, \*eph;

ph = (struct proghdr \*)((uintptr\_t)ELFHDR + ELFHDR->e\_phoff);

eph = ph + ELFHDR->e\_phnum;

for (; ph < eph; ph ++) {

readseg(ph->p\_va & 0xFFFFFF, ph->p\_memsz, ph->p\_offset);

}

((void (\*)(void))(ELFHDR->e\_entry & 0xFFFFFF))();

bad:

outw(0x8A00, 0x8A00);

outw(0x8A00, 0x8E00);

while (1);

}

readsect从设备的第secno扇区读取数据到dst位置

static void

readsect(void \*dst, uint32\_t secno) {

waitdisk();

outb(0x1F2, 1); *// 设置读取扇区的数目为1*

outb(0x1F3, secno & 0xFF);

outb(0x1F4, (secno >> 8) & 0xFF);

outb(0x1F5, (secno >> 16) & 0xFF);

outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);

*// 上面四条指令联合制定了扇区号*

*// 在这4个字节线联合构成的32位参数中*

*// 29-31位强制设为1*

*// 28位(=0)表示访问"Disk 0"*

*// 0-27位是28位的偏移量*

outb(0x1F7, 0x20); *// 0x20命令，读取扇区*

waitdisk();

insl(0x1F0, dst, SECTSIZE / 4); *// 读取到dst位置，*

*// 幻数4因为这里以DW为单位*

}

IO地址 功能   
0x1f0 读数据，当0x1f7不为忙状态时，可以读。   
0x1f2 要读写的扇区数，每次读写前，你需要表明你要读写几个扇区。最小是1个扇区   
0x1f3 如果是LBA模式，就是LBA参数的0-7位   
0x1f4 如果是LBA模式，就是LBA参数的8-15位   
0x1f5 如果是LBA模式，就是LBA参数的16-23位   
0x1f6 第0~3位：如果是LBA模式就是24-27位 第4位：为0主盘；为1从盘   
0x1f7 状态和命令寄存器。操作时先给命令，再读取，如果不是忙状态就从0x1f0端口读数据

加载ELF文件

bootmain(void) {

..........

if (ELFHDR->e\_magic != ELF\_MAGIC) {*//这里有个判断*

goto bad;

}

struct proghdr \*ph, \*eph;

*//ELF头部有描述ELF文件应加载到内存什么位置的描述表，这里读取出来将之存入ph*

ph = (struct proghdr \*)((uintptr\_t)ELFHDR + ELFHDR->e\_phoff);

eph = ph + ELFHDR->e\_phnum;

*//按照程序头表的描述，将ELF文件中的数据载入内存*

for (; ph < eph; ph ++) {

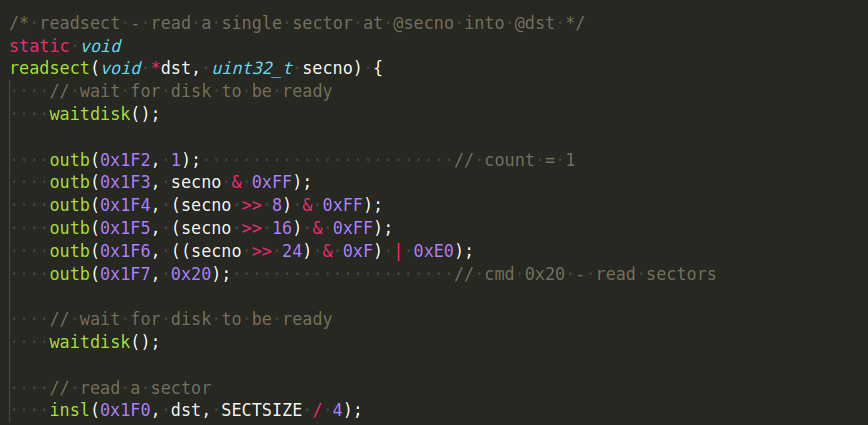
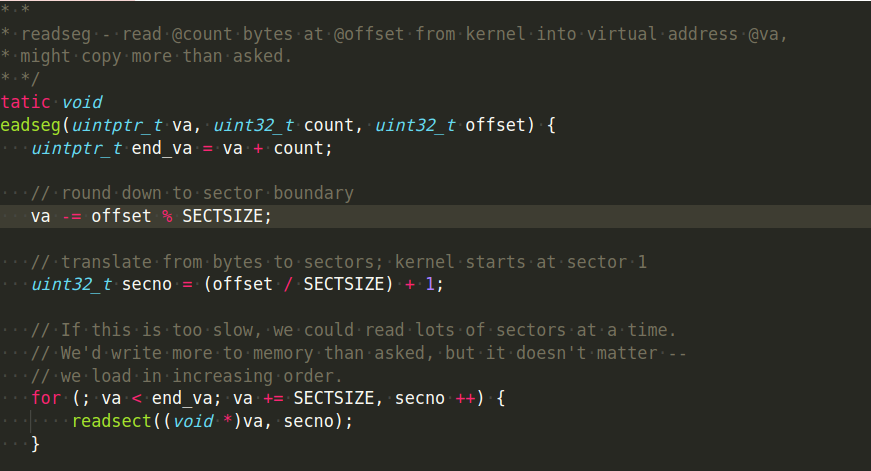
readseg(ph->p\_va & 0xFFFFFF, ph->p\_memsz, ph->p\_offset);

((void (\*)(void))(ELFHDR->e\_entry & 0xFFFFFF))();*//根据ELF头表中的入口信息，找到内核的入口并开始运行*

bad:

..........

}



# 练习5

题目：   
实现函数调用堆栈跟踪函数

### 什么是函数栈？

当函数被调用时，首先会把函数的参数依次入栈（这里指的是堆栈传参，当然也可以用寄存器传参）调用函数的栈底压栈到自己函数的栈中（push bp），然后将原来函数栈顶sp作为当前函数的栈底（mov bp,sp）。函数运行完成时，会将压入栈中的bp重新出栈到bp中（pop bp）。同时将计算的结果保存在寄存器中，返回原界面。

那么我们可以粗浅的建立一个栈模型

ss:[ebp-8] ;变量2ss:[ebp-4] ;变量1ss:[ebp] ;栈针ss:[ebp+4] ;返回地址ss:[ebp+8] ;第一个参数

### 函数实现

read\_ebp()和read\_eip()函数来获取当前ebp寄存器和eip 寄存器的信息。

查看print\_stackframe函数注释

/\* LAB1 YOUR CODE : STEP 1 \*/

/\* (1) call read\_ebp() to get the value of ebp. the type is (uint32\_t);

\* (2) call read\_eip() to get the value of eip. the type is (uint32\_t);

\* (3) from 0 .. STACKFRAME\_DEPTH

\* (3.1) printf value of ebp, eip

\* (3.2) (uint32\_t)calling arguments [0..4] = the contents in address (unit32\_t)ebp +2 [0..4]

\* (3.3) cprintf("\n");

\* (3.4) call print\_debuginfo(eip-1) to print the C calling function name and line number, etc.

\* (3.5) popup a calling stackframe

\* NOTICE: the calling funciton's return addr eip = ss:[ebp+4]

\* the calling funciton's ebp = ss:[ebp]

\*/

for(i = 0; ebp!=0 && i < STACKFRAME\_DEPTH; i++) {*//STACKFRAME\_DEPTH = 20 一直向上循环找到所有的调用函数为止，一开始没有判断栈针为空的条件*

cprintf("ebp:0x%08x eip:0x%08x ",ebp, eip);

uint32\_t \*args = (uint32\_t \*)ebp + 2; *//传参*

for(j = 0; j < 4; j++)

cprintf("0x%08x ",args[j]);

cprintf("\n");

print\_debuginfo(eip-1);

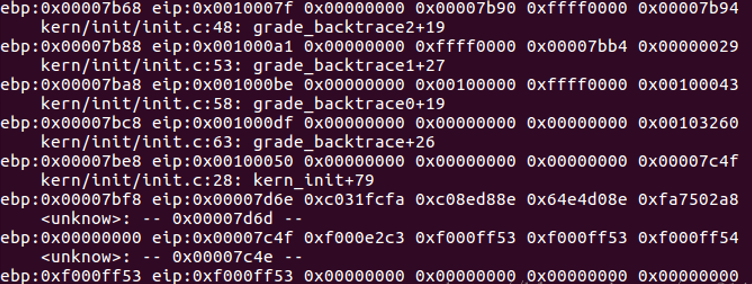
*//模拟函数执行完毕*

eip = \*((uint32\_t \*)ebp+1);*//调用函数的返回地址*

ebp = \*((uint32\_t \*)ebp);*//上一个函数的栈针 //循环直到没有调用函数停止*

}

执行结果如下：   
未加ebp!=0



我们发现ebp的值在0x7bf8之后就为零了，这说明上面没有了调用函数，直接加个判断   
ebp!=0 就可以输出预期的结果。

加了ebp!=0

# 练习6

题目：   
1.中断向量表中一个表项占多少字节？其中哪几位代表中断处理代码的入口？   
2.请编程完善kern/trap/trap.c中对中断向量表进行初始化的函数idt\_init。在idt\_init函数中，依次对所有中断入口进行初始化。使用mmu.h中的SETGATE宏，填充idt数组内容。注意除了系统调用中断(T\_SYSCALL)以外，其它中断均使用中断门描述符，权限为内核态权限；而系统调用中断使用异常,权限为陷阱门描述符。每个中断的入口由tools/vectors.c生成，使用trap.c中声明的vectors数组即可。   
3.请编程完善trap.c中的中断处理函数trap在对时钟中断进行处理的部分填写trap函数中处理时钟中断的部分，使操作系统每遇到100次时钟中断后，调用 print\_ticks子程序，向屏幕上打印一行文字100 ticks。

### 1.中断向量表中一个表项占多少字节？其中哪几位代表中断处理代码的入口？

中断向量表一个表项占用8字节，其中2-3字节是段选择子，0-1字节和6-7字节拼成位移，   
两者联合便是中断处理程序的入口地址。   
中断描述符表中一个表项占8个字节 ，其中015和4863分别为offset的低16位和高16位，16~31位是段选择子，通过段选择子得到段基址，再加上段内偏移量就可以得到中断处理代码的入口。

### 2.请编程完善kern/trap/trap.c中对中断向量表进行初始化的函数idt\_init

extern uintptr\_t \_\_vectors[];*//声明\_\_vertors[] You can use "extern uintptr\_t \_\_vectors[];" to define this extern variable which will be used later.*

int i;

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

SETGATE(idt[i],0,GD\_KTEXT,\_\_vectors[i],DPL\_KERNEL);*//对整个idt数组进行初始化*

}

SETGATE(idt[T\_SWITCH\_TOK],0,GD\_KTEXT,\_\_vectors[T\_SWITCH\_TOK],DPL\_USER);*//在这里先把所有的中断都初始化为内核级的中断*

lidt(&idt\_pd);*//使用lidt指令加载中断描述符表 just google it! and check the libs/x86.h to know more.利用google找到了相关函数*

}*/\**

*传入的第一个参数gate是中断的描述符表*

*传入的第二个参数istrap用来判断是中断还是trap*

*传入的第三个参数sel的作用是进行段的选择*

*传入的第四个参数off表示偏移*

*传入的第五个参数dpl表示这个中断的优先级*

*\*/*

### 3.程完善trap.c中的中断处理函数trap在对时钟中断进行处理的部分填写trap函数中处理时钟中断的部分，使操作系统每遇到100次时钟中断后，调用 print\_ticks子程序，向屏幕上打印一行文字100 ticks

实验代码填写

case IRQ\_OFFSET + IRQ\_TIMER:

/\* LAB1 YOUR CODE : STEP 3 \*/

/\* handle the timer interrupt \*/

/\* (1) After a timer interrupt, you should record this event using a global variable (increase it), such as ticks in kern/driver/clock.c

\* (2) Every TICK\_NUM cycle, you can print some info using a funciton, such as print\_ticks().

\* (3) Too Simple? Yes, I think so!

\*/

代码如下：

ticks++;

if(ticks%TICK\_NUM == 0)*//每次时钟中断之后ticks就会加一 当加到TICK\_NUM次数时 打印并重新开始*

print\_ticks();*//前面有定义 打印字符串*

实验截图如下：

