# lab1 Report

2016011398 高鸿鹏

## 练习1.1：操作系统镜像文件ucore.img是如何一步一步生成的?

通过注释找到`ucoreimg`创建的语句为：

```
UCOREIMG    := $(call totarget,ucore.img)

$(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)
```

可以看到`ucore.img`的创建需要kernel和bootblock，再寻找创建kernel和bootblock的语句：

### 1. 创建kernel的语句：

```
    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)
```

创建kernel需要准备init.o，readline.o，studio.o，kdebug.o，kmonitor.o，panic.o，clock.o，console.o，intr.o，picirq.o，trap.o，trapentry.o，vector.o，pmm.o，printfmt.o，string.o，其中trapentry.o和vector.o由`.S`生成，其余均由`.c`文件生成。上述代码段最后一句话生成kernel文件。

这里罗列init.o和vector.o以及kernel文件生成的代码

```
    #Makefile当中的宏批量产生代码如下
    $(call add_files_cc,$(call listf_cc,$(KSRCDIR)),kernel,$(KCFLAGS))

    #生成init.o的实际代码如下
    gcc -Ikern/init/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-
stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/
-c kern/init/init.c -o obj/kern/init/init.o

    #生成vector.o的代码
    gcc -Ikern/trap/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-
stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/
-c kern/trap/vectors.S -o obj/kern/trap/vectors.o
```

```
9
10    #生成kernel的代码
11    ld -m    elf_i386 -nostdlib -T tools/kernel.ld -o bin/kernel
12    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
13
14    # 其中的关键参数 -T<scriptfile> 规定连接器使用指定脚本
```

## 2. 创建bootblock的语句：

```
1     bootfiles = $(call listf_cc,boot)
2     $(foreach f,$(bootfiles),$(call cc_compile,$(f),$(CC),$(CFLAGS) -Os -
      nostdinc))
3
4     bootblock = $(call totarget,bootblock)
5
6     $(bootblock): $(call toobj,$(bootfiles)) | $(call totarget,sign)
7      @echo + ld $@
8      $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ -o $(call
      toobj,bootblock)
9      @$(OBJDUMP) -S $(call objfile,bootblock) > $(call asmfile,bootblock)
10     @$(OBJCOPY) -S -O binary $(call objfile,bootblock) $(call
      outfile,bootblock)
11     @$(call totarget,sign) $(call outfile,bootblock) $(bootblock)
12
13    $(call create_target,bootblock)
14
```

能够发现bootblock的生成需要sign和bootfiles文件，通过最后一句话生成bootbolck文件

### 2.1 bootfiles的生成

```
1  bootfiles = $(call listf_cc,boot)
2  $(foreach f,$(bootfiles),$(call cc_compile,$(f),$(CC),$(CFLAGS) -Os -
   nostdinc))
```

- bootfiles的需要bootasm.o和bootmain.o来生成
- 在Makefile当中对文件bootasm与bootmain文件的创建是宏批量产生的
- 创建bootasm的代码实际为：

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

- 创建bootmain的代码实际为：

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

- 相关参数解释如下：

```
-ggdb：尽可能的生成gdb的可以使用的调试信息，并使用gdb+qemu的模式对ucore操作系统或者bootloader进行调试。

-m32：加上该参数之后生成32位的代码，因为qemu模拟的是32位的80386，所以操作系统也需要是32位

-gstabs：生成stabs的调试信息。可以方便开发者阅读ucore的monitor生成的函数调用栈信息

-nostdinc：不使用标准库，因为编写的是ucore操作系统，其内核应该自给自足，不能借用东西

-fno-stack-protector：禁用栈保护机制。

-Os：为了减小代码长度而进行优化，因为缓冲区只有512字节，最后两个字节还要用作结束标志55AA，所以要尽可能减小代码长度

-I<direction> 表示在direction的目录下进行搜索或者进行操作

-fno-builtin：不适用C语言的内建函数。
```

### 2.2 sign的生成

```
# create 'sign' tools
$(call add_files_host,tools/sign.c,sign,sign)
$(call create_target_host,sign,sign)
```

用sign.c创建了sign.o，再用sign.o创建sign文件

### 2.3 用bootasm.o和bootmain.o生成bootblock.o

```
# 完整代码如下：
ld -m    elf_i386 -nostdlib -N -e start -Ttext 0x7C00 \
  obj/boot/bootasm.o obj/boot/bootmain.o -o obj/bootblock.o

# 对其中关键参数的解释

-m 模拟为80386上的连接器
-nostdlib 不使用标准库
-N 设置数据段和代码段均可读写
-e start指定代码段入口
-Ttext 指定代码段开始的地址
这段代码表示使用bootmain.o和bootasm.o创建bootblock.o文件
```

### 2.4 用bootblock.o生成bootblock.out

```
objcopy -S -O binary obj/bootblock.o obj/bootblock.out

其中的关键参数为

-S 移除所有符号与重定位信息
```

## 3. 其他重要代码

```
$(V)dd if=/dev/zero of=$@ count=10000
$(V)dd if=$(bootblock) of=$@ conv=notrunc
$(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc

#以下为代码解释
    1. 创建一个有10000块的文件，并且在默认初始化的时候文件中的每一个块中数据为0
    2. 将第一个块中的内容替换为bootblock的数据
    3. 从第二个块开始往里面写kernel的数据
```

## 练习1.2：一个被系统认为是符合规范的磁盘主引导扇区的特征是什么

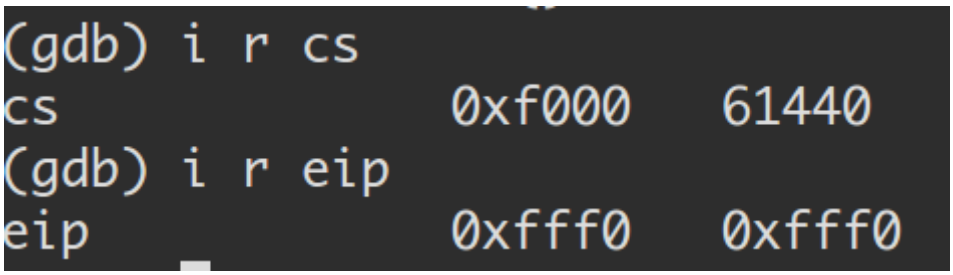
1. 一个块的到校为512字节，并且最后两个字节必须是55AA
2. 文件内容的总大小不超过510字节

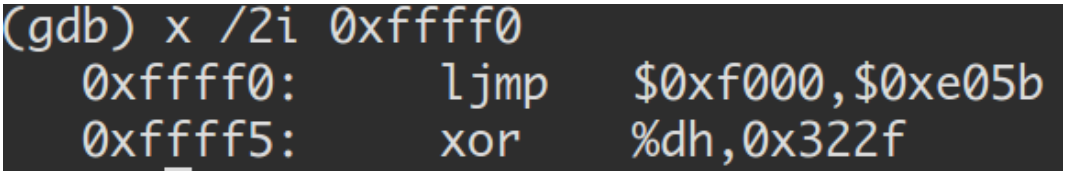
## 练习2：使用qemu调试执行lab1当中的软件(要求在实验报告中简单写出实验过程)

1. **从CPU加电执行的第一条命令开始，单步追踪BIOS的执行**
    - 首先修改tools/gdbinit当中的内容，并在lab1的目录下执行make debug

    ```
    set architecture i8086      //设置当前调试的CPU为80386
    target remote :1234         //链接qemu，并让qemu进入等待模式，听从gdb的命令
    ```

    - 用相关指令查看cs和eip的初值如图,通过公式`address=cs*16+eip`得到第一条指令的地址在0xffff0处，再通过命令`x /2i 0xffff0`查看从0xffff0开始的两条指令如图所示，可以看到第一条指令是一个长跳指令，并且第一条指令存储在`cs||eip = 000e05b`的位置，可以用si的命令单步执行BIOS。

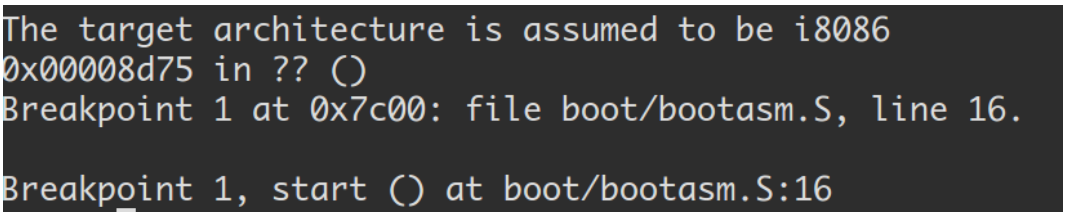
    
    

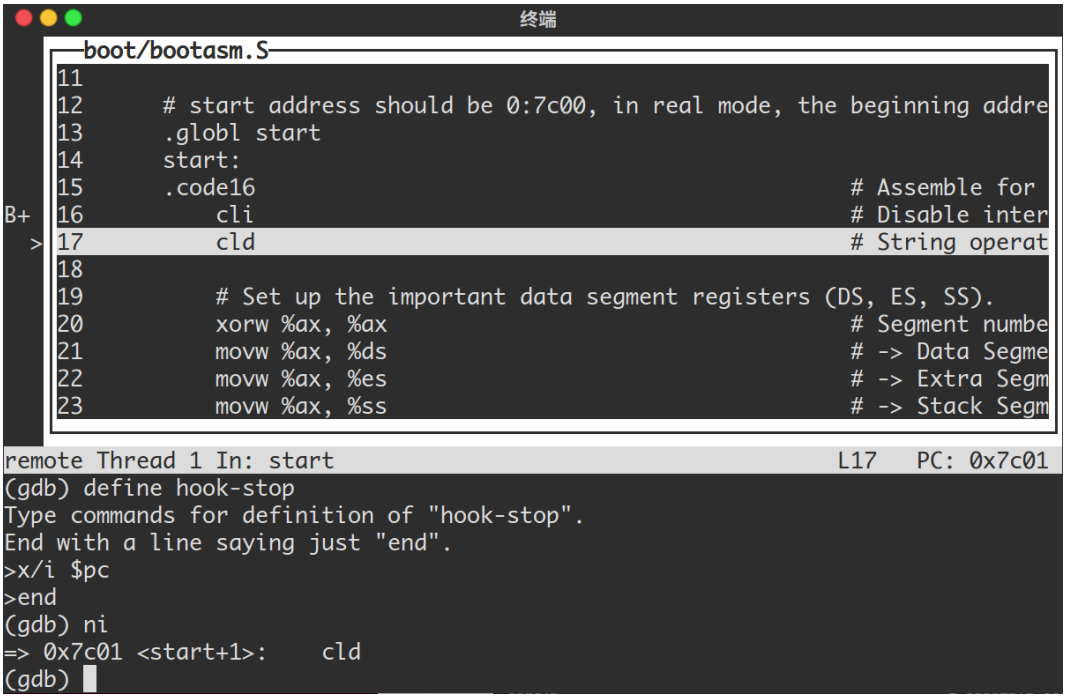
    
    

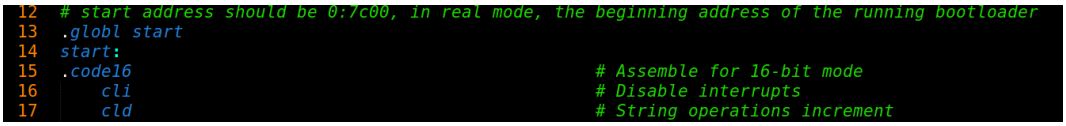
2. **在初始化位置0x7c00设置实地址断点,测试断点正常。从0x7c00开始跟踪代码运行,将单步跟踪反汇编得到的代码与bootasm.S和 bootblock.asm进行比较。**
    - 首先修改gdbinit文件，将文件当中的内容改为以下代码，然后make debug

```
file obj/bootblock.o
set architecture i8086
target remote :1234
b *0x7c00
continue
```

- 发现gdb中的显示如图，表示程序的入口地址为`bootasm.S`中的第16行，并且在0x7c00成功设置断点，同时发现在显示窗口里面程序停在了`bootasm.S`的第16行，此时用实验指导书的gdb强制反汇编当前指令之后，单步执行一下，得到当前指令为cld，则说明断点正常。从第四张图片和第二张图片可以看得出来`bootasm.S`和`bootclock.asm`当中的值是一样的.



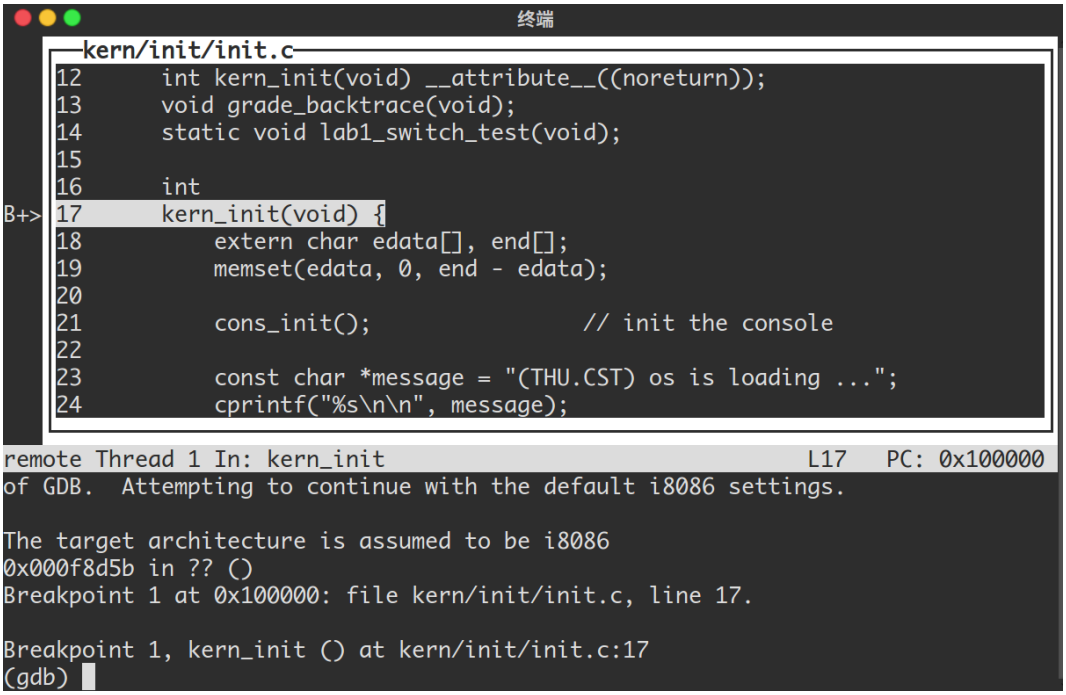




3. **自己找一个bootloader或内核中的代码位置，设置断点并进行测试。**

- 找到kernel的一个组成部分init文件，在其中找到一个函数kernel_init()，在这里设置一个断点，然后修改gdbinit文件，再执行make debug，再用ni指令单步执行，测试完成。

```
file bin/kernel
set architecture i8086
target remote :1234
b *kern_init
continue
```




## 练习3：分析bootloader进入保护模式的过程。（要求在报告中写出分析）

- 初始化环境，初始化所有的段寄存器，为开启A20做准备

```
.code16
        cli
        cld
        xorw %ax, %ax
        movw %ax, %ds
        movw %ax, %es
        movw %ax, %ss
```

- 开启A20，以及如何开启A20

  如果不开启A20，在使用32位地址线的时候，系统只能访问奇数兆的内存。为了在保护模式下访问所有的内存(32位),A20端口必须开启。

  两段程序的前两段都是先检查端口0x64是否空闲，如果有缓存就再次检查。第一段中发现0x64端口空闲之后，发送0xd1命令向8042's P2 port写数据，即禁止键盘操作。在第二段中，等待0x64端口空闲，再将8042芯片的输出端口的第一个bit置1即可，到此完成A20地址线的激活。

```
seta20.1:
    inb $0x64, %al
    testb $0x2, %al
    jnz seta20.1
    movb $0xd1, %al
    outb %al, $0x64

seta20.2:
    inb $0x64, %al
    testb $0x2, %al
    jnz seta20.2

    movb $0xdf, %al
    outb %al, $0x60
```

- 初始化GDT表，并进入保护模式

  将gdtdesc当中的值导入到GDTR当中，可以看到gdtdesc只想的地址空间已经有了一个简单的全局描述符表，然后将cr0寄存器的最后一位置1，使能保护模式，到此进入保护模式

```
    lgdt gdtdesc
    movl %cr0, %eax
    orl $CR0_PE_ON, %eax
    movl %eax, %cr0

    gdtdesc:
    .word 0x17                                      # sizeof(gdt) - 1
    .long gdt                                       # address gdt
```

- 跳转到下一条指令，这时已经在32位环境下了，在这种情况下更新CS的基地址，开始执行32位代码

```
ljmp $PROT_MODE_CSEG, $protcseg
```

- 在32位的保护模式下对段寄存器进行初始化，并且建立堆栈指针，将栈的七点设为start(0x7c00)

```
protcseg:
    # Set up the protected-mode data segment registers
    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
```

- 最后在进入保护系统之后调用bootmain

```
    call bootmain
```

## 练习4：分析bootloader加载ELF格式的OS的过程。（要求在报告中写出分析）

- waitdisk(void)函数等待磁盘准备好
- redsect函数,在磁盘准备好之后，首先设置要读写1个扇区，在LBA模式下，0x1F3是32位参数当中的0-7位，0x1F4是8-15位，0x1F5是16-23位，0x1F6中存24-31位，其中的28-31位的内容直接为0111，28位的0表示访问的是Disk 0，0-27是访问的Offset。到此已经完成对访问设备的第secno扇区的初始化准备工作，0x20指令给出0x20（读取）的指令，然后在磁盘准备好之后，将磁盘扇区的数据读到dst指向的内存当中。
- redsect函数的作用就是将磁盘的第secno号扇区的内容读到dst指向的位置。

```
static void readsect(void *dst, uint32_t secno) {
    waitdisk();

    outb(0x1F2, 1);
    outb(0x1F3, secno & 0xFF);
    outb(0x1F4, (secno >> 8) & 0xFF);
```

```
    outb(0x1F5, (secno >> 16) & 0xFF);
    outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
    outb(0x1F7, 0x20);

    waitdisk();

    insl(0x1F0, dst, SECTSIZE / 4);
}
```

- redseg函数是对redsect()函数的包装，使得通过该函数能够从扇区secno当中读取任意长度的字符。在初始化secno的时候因为0扇区当中保存的是BIOS，所以从1扇区开始读

```
static void readseg(uintptr_t va, uint32_t count, uint32_t offset) {
    uintptr_t end_va = va + count;

    va -= offset % SECTSIZE;

    uint32_t secno = (offset / SECTSIZE) + 1;

    for (; va < end_va; va += SECTSIZE, secno ++) {
        readsect((void *)va, secno);
    }
}
```

- bootmain函数，首先读取磁盘的第一个扇区（也就是ELF头部），通过储存的e_magic变量确定ELF文件合法，再将储存在ELF头部的描述ELF文件加载到内存指定地址的描述表的头地址储存在ph中，再用redseg函数将ELF程序段加载到内存中（ELF文件0x1000位置后面的0xd1ec比特被载入内存0x00100000，0xf000位置后面的0x1d20比特被载入内存0x0010e000），载入程序段结束后，再从ELF头部存储的e_entry信息找到kernel的入口，并且没有返回值，至此bootloader已经完成对ELF格式的ucore操作系统的加载。

```
void 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);
}
```

## 练习5：实现函数调用堆栈跟踪函数 （需要编程）

添加的函数如下，需要注意了解cprintf的用法，在阅读了函数堆栈之后，了解了局部变量，栈顶，栈底寄存器的位置关系之后，再参考print_stackframe函数的注释之后完成对函数的补全，具体代码如下：

```
void
print_stackframe(void) {
     /* LAB1 2016011398 : STEP 1 */
    uint32_t my_ebp = read_ebp();
    uint32_t my_eip = read_eip();
    int i,j;
    for(i = 0;/*ebp!=0 && */i < STACKFRAME_DEPTH;i ++){
        cprintf("ebp = 0x%08x ,eip = 0x%08x\n",ebp,eip);
        uint32_t *args = (uint32_t *)ebp + 2;
        for(j = 0;j < 4;j ++){
            cprintf("args[0x%08x] = 0x%08x ",j,args[j]);
        }
        cprintf("\n");
        print_debuginfo(eip-1);
        eip = ((uint32_t *)ebp)[1];
        ebp = ((uint32_t *)ebp)[0];
    }
}
```

其中最后一行输出如下：

```
ebp:0x00007bf8 ,eip:0x00007d6a args: 0xc031fcfa  0xc08ed88e  0x64e4d08e
0xfa7502a8
    <unknow>: -- 0x00007d69 --
```

在最后一行中是第一个使用堆栈的函数bootmain.c当中的bootmain函数，bootmain函数的起始地址被规定为0x7c00，所以在call bootmain之后，此时的栈底寄存器中的值为0x7b68

## 练习6：完善中断初始化和处理 （需要编程）

1. **中断描述符表（也可简称为保护模式下的中断向量表）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？**

   中断描述符表中的每一个表项占8字节，0-1字节和6-7字节拼接起来形成Offset，2-3字节是段选择子，根据段选择子在段描述符表中找到基址。

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。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。**

   代码中添加的函数如下，根据注释第一条为"extern uintptr_t __vectors[];",第二步完成对中断描述符表中所有表项的初始化，第三步讲控制权从用户态转到内核态，因为lidt指令只能在内核态执行，最后用lidt指令加载中断描述符表

```
void
idt_init(void) {
      extern uintptr_t __vectors[];
      /* setup the entries of ISR in IDT*/
      int i;
      /*#define SETGATE(gate, istrap, sel, off, dpl)*/
      for(i = 0;i<sizeof(idt)/sizeof(struct gatedesc); i++){
            SETGATE(idt[i] ,0 ,GD_KTEXT ,__vectors[i],DPL_USER);
      }
      SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, __vectors[T_SWITCH_TOK], DPL_USER);
      lidt(&idt_pd);
}
```

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”。**

   添加的代码如图，根据提示，先将记录时间的全局变量+1，然后每遇到100次始终终端的时候，就调用print_ticks函数。

```
ticks++;
if(ticks % TICK_NUM == 0){
    print_ticks();
  }
```