# **练习1：理解通过make生成执行文件的过程。**

生成bin/kern部分  
生成init.o  
+ cc kern/init/init.c  
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readline.  
+ cc kern/libs/readline.c  
gcc -Ikern/libs/ -fno-builtin -wall-ggdb -m32-gstabs -nostdinc -fno-stack protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/ libs/readline.c -o obj/kern/libs/readline.o  
  
生成stdio.d  
+ cc kern/Libs/stdio.c  
gcc -Ikern/Libs/-fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/ Libs/stdio.c-oobj/kern/ibs/stdio.o

生成kdebug.o  
+ cc kern/debug/kdebug.c  
gcc-Ikern/debug/ -fno-builtin -wall -ggdb -m32 -gstabs -nostdinc -fno-stack protector -Iibs/-Ikern/debug/-1kern/dciver/ -Ikern/trap/ -Ikern/mm/ -c kern/ debug/kdebug.colobi/kern/debug/kdebug.o  
生成kmonitor.o  
+ cc kern/debug/kmonitor.c  
gcc -Ikern/debug/ -fno-builtin -Wall -ggdb-m32 -gstabs -nostdinc -fno-stack protector ILibs/ -Ikern/debug/-Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kerny debug/kmonitor.c -oobj/kern/debua/kmonitor.o

生成panic.o  
+ cc kern/debug/panic.c  
cc -Ikern/debug/-fno-builtin -walL-ggdb -m32 -gstabs nostdinc fno-stack protector -ILibs/ Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/-c kern/ debug/panic.c -o obj/kern/debug/panic.o

生成clock.o  
+ cc kern/driver/clock.c  
gcc -Ikern/driver/ -fno-butLtin-wall -ggdb -m32 -gstabs -nostdtnc -fno-stack protector -Ilibs/-Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/ driver/clock.c-oobj/kern/driver/clock.o

生成console.o  
+ cc kern/driver/console.c  
gcc -Ikern/driver/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/ driver/console.c -o obj/kern/driver/console.o

生成intr.o  
+ cc kern/driver/intr.c  
gcc -Ikern/driver/ -fno-builtin -wall -ggdb -m32 -gstabs -nostdinc -fno-stack protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/-c kern/ driver/intr.c -o obj/kern/driver/intr. O

生成picirq.o  
+ cc kern/driver/picirq.c  
gcc -Ikern/driver/ -fno-builtin -wall-ggdb -n32-gstabs-nostdinc -fno-stack protector -Ilibs/ Ikern/debug/ -Ikern/driver/-Ikern/trap/ -Ikern/mn/ -c kern/ driver/picirq.c-olobj/kern/driver/picirq.o

生成trap.o  
+ cc kern/trap/trap. C  
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/trap.co obj/kern/trap/trap.o   
生成trapentry.o

+ cc kern/trap/trapentry.s  
gcc -Ikern/trap/ -fno-builtin -WalL -ggdb -m32 -gstabs -nostdinc -fno-stack protector -Ilibs/-Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mn/ -c kern/ trap/trapentry.so obj/kern/trap/trapentry-o  
生成vectors.o

+ cc kern/trap/vectors.s  
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

生成pmm.o  
+ cc kern/m/pmm.c  
gcc -Ikern/mm/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack protector -Ilibs/ -1kern/debug/-Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/ nm/pmm.c-o obj/kern/mm/pmm.o  
솔

生成printfmt.o  
+ cc Libs/printfmt.cl  
gcc -Ilibs/-fno-builtin-wall -ggdb -m32 -gstabs -nostdinc -fno-stack protector -Ilibs/ -c Libs/printfnt.c-o obj/Libs/printfmt.o

生成string.o  
+ cc libs/string.c  
gcc-Ilibs/ -fno-builtin-Wall -ggdb -m32-gstabs -nostdinc -fno-stack protector -Ilibs/-c libs/string.c o obj/libs/string.o

用上面.o文件连接生成bin/kern  
+ Id bin/kernel  
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/traNtrapentry.o obj/kern/trap/vectors.o obj/kern/mm/ pmm.o obj/ibs/printfmt.o obj/libs/string.o

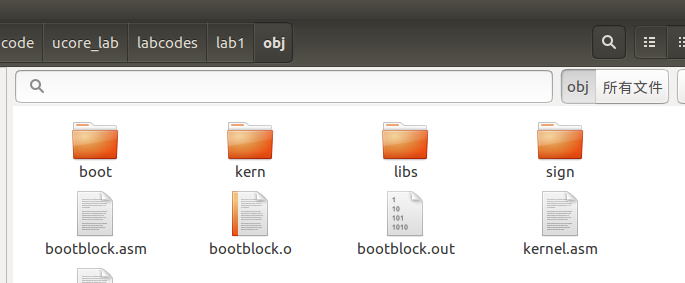
生成bin/bootblock部分  
生成bootasm.o  
+ cc boot/bootasm.s  
gcc -Iboot/-fno-butltin -wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Os nostdinc -c boot/bootasm.s -o obj/boot/bootasnm.o

生成bootmain.o  
+ cc boot/bootmatn.c  
gcc -Iboot/ -fno-butltin -Wall -ggdb -m32-gstabs -nostdinc -fno-stack-protector -ILbs/ -Os -nostdinc -c boot/bootmain.c -o obj/boot/bootmatn.o

用上面两个.0文件生成bin/bootblock

生成bin/sign  
生成sign.o  
+ cc tools/sign.c  
gcc -Itools/ -g-Wall -02 -c tools/sign.c -o obj/sign/tools/sign.o gcc -g -wall -02 obj/sign/tools/sign.o -o bin/sign  
用上面.o生成bin/sign

生成 ucore.img  
dd if=/dev/zero of-bin/ucore.img count=10000 dd if=bin/bootblock of=bin/ucore.img conv=notrunc dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc



## 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？

主引导扇区大小为512字节，并且最后结尾为 0x55，0xA

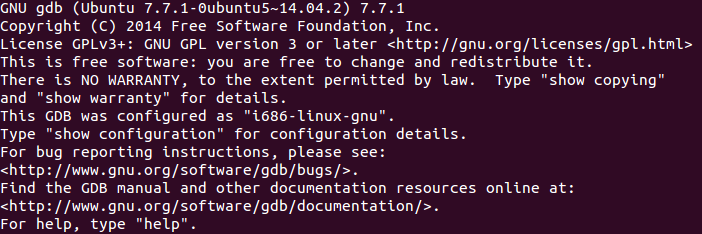
# 练习2：使用qemu执行并调试lab1中的软件

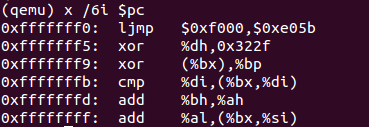
1 首先启动一个终端，输入： qemu -S -s -hda bin/ucore.img -monitor stdio 的方式启动qemu，等待gdb连接





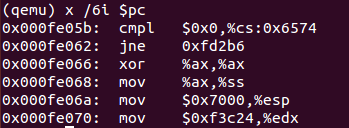
2 在启动一个终端，启动gdb用远程连接的方式连接qemu

3 在qemu终端输入 x /6i $pc 这条指令，观察qemu启动时候首先执行的指令



可以看到现在的CS:IP --> 0xffff0，这是指向内存的开始位置，也是加电后bios初始化后cs:ip的位置。

然后在gdb终端输入si执行一条汇编指令，然后在qemu终端观察下面的执行指令已经变为下面

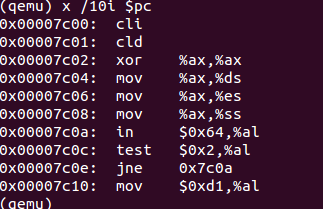


也就是第一条指令后跳转后的位置

## 2.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。

使用si单步执行观察下面10条指令，进行比较。

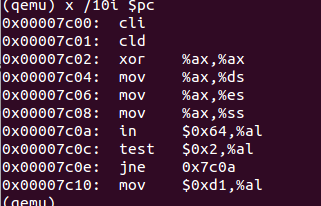
得到如下结果：



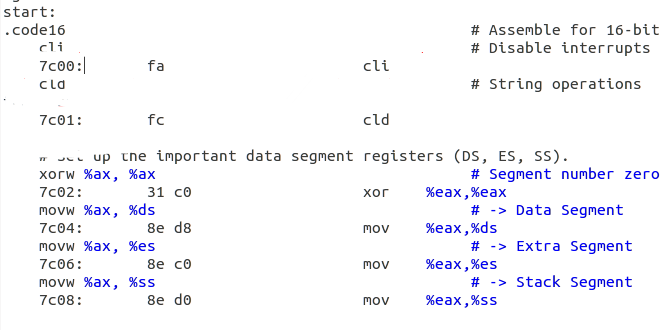
## 3.从0x7c00开始跟踪代码运行,将单步跟踪反汇编得到的代码与bootasm.S和 bootblock.asm进行比较。

使用si单步执行观察下面10条指令，进行比较。

得到的代码是：

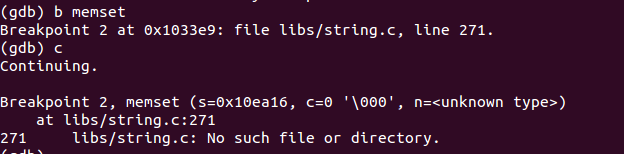


Bootlock.asm代码是：



## 4.找到bootloader的代码位置，进行测试。

在gdb终端载入file ./bin/kernel,然后设置暂停点memset，用c执行，得到下面结果：



# 练习3

。。。。

练习4

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); 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); }

加载ELF文件

bootmain(void) {

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))();

}

练习5：实现函数调用堆栈跟踪函数

print\_stackframe(void) {

uint32\_t ebp = read\_ebp(), eip = read\_eip();

int i, j;

for (i = 0; ebp != 0 && i < STACKFRAME\_DEPTH; i ++) {

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

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

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

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

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

}

cprintf("\n");

print\_debuginfo(eip - 1);

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

eip = ((uint32\_t \*)ebp)[1];

ebp = ((uint32\_t \*)ebp)[0];

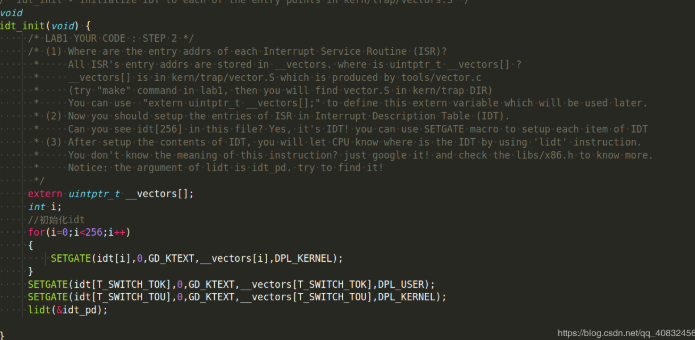
//popup a calling stackframe

}

练习6：

(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。



(2) 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[];

\_\_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);

}

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

}

### 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

ticks++; if(ticks%TICK\_NUM == 0)

print\_ticks();

实验图

