OSlab1

组员：

1811437 吴昌昊

1811491 林正青

1811515 於一帆

分工：

吴昌昊（exe3、5、challenge1）

林正青（exe2、6）

於一帆（exe1、4、challenge2）

遇到的坑和反思

以下文档记录了本小组在完成实验过程中除了技术和原理理解之外遇到的疑问。在经过思考和讨论后，我们对这些疑问给出了自己的解答，也对实验的实现有了更深刻的理解。

练习1

Q：bootblock与其他文件的逻辑是什么?

A：起初，我以为每次的操作系统加载都需要执行makefile的过程生成bootblock和ucore.img等文件，但是在思考后，做出了想法的更正：我们这次的练习是一个已知makefile去反推构成的过程，与实际的操作系统加载和处理是相反的。也就是说，我们用makefile的逻辑去生成了一个os的主引导扇区和镜像，它们可以被用在一台裸机上，电脑在载入os时，直接载入这个主引导扇区和镜像即可，其中已经包含了我们写在里面的代码逻辑。

Q：sign的作用是什么?

A：起初我一直以为sign也是主引导扇区的组成部分，所以分析代码的时候一直不理解它和主引导扇区功能是怎么对应的。后来发现，sign是将bootasm.s(实现保护模式转换)和bootmain.c(往内存读kernel)结合形成的bootblock.out转换成了一个512字节的符合主引导扇区要求的主引导扇区bootblock。也就是说，sign是一个包装的工具。

练习2

Q：在gdbinit文件中加入set architechture i8086和break *0xffff0两条命令，尝试查看BIOS执行的第一条指令时，gdb调试失败，卡在如下位置，没有进入命令行模式，显然gdbinit文件修改得并不正确。到底应当给出什么命令?

A：按照指导书中写法，将gdbinit文件内容改成只有那两行命令即可。如果按照原来gdbinit文件里的写法，在kern_init函数入口处停下，地址则是0x100000H，但kern_init函数本身用于加载操作系统内核，与BIOS无关。

练习3

Q：A20的“线号乌龙“

A：一开始查资料的时候一直看到A20是在第20根线上完成的兼容开关设置，我很疑惑，原来就是20根线，现在却在第20根线上做开关，那不是占用了原本那根线的传输功能么？后来再找了一些资料求证，可以合理认为，这里的”第20根“是从”0“开始编号计数的。开关是放在了实际增加的第1根线上，用于控制后面的地址线传输是否有效。

练习4

Q：readseg函数里的"va -= offset % SECTSIZE;"是什么用途？

A：因为readseg每次读入的都是扇区大小的整数倍，读进内存以后va和实际要读的区域入口有一段距离，用va减去这个空出的值得到va‘，这时读入的扇区放到va‘开始的地方，则正好使要读入的地方和原来的va对其。

Q：readseg函数里"uint32_t secno = (offset / SECTSIZE) + 1;"为什么要+1？

A：因为0号扇区是主引导扇区，要避免再加载一遍。

Q：bootmain函数最后跳去了哪里？为什么要跳？

A：使用gdb查看代码，发现跳到了0x00100000的位置，即kern_init的函数入口处，也就是操作系统在内存中被载入的位置。

练习5

Q：堆栈最下层的eip是什么?

A：是bootasm跳到bootmain后，保存的返回地址。

详见bootasm.s的代码部分：

```
// Set up the stack pointer and call into C. The stack region is from 0--start(0x7c00)
   movl $0x0, %ebp
   movl $start, %esp
   call bootmain
```

Q：为什么ebp是7bf8?

A：因为bootasm在call bootmain之后，会从7c00开始，把返回地址和原本的ebp压栈，使得现在的ebp变成了7c00-8-8=7bf8。

Q：整个查看堆栈的结果是按照什么逻辑显示出来的?

A：运行kdebug.c中的print_stackframe的函数后，函数会在0-7c00的栈中，从低位向高位扫描，读出其中的ebp和eip。越先打印出来的，就是约在栈顶的ebp，也就是越后执行的函数。从打印出来的信息可以看出函数的调用顺序是：bootasm.s -> bootmain (->) kern_init ->grade_backtrace -> grade_backtrace1 -> grade_backtrace2 -> mon_backtrace -> print_stackframe

Q：为什么没有bootmain到kern_init的函数栈帧切换?

A：查看bootmain的最后， ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();可见是使用指针跳转到了kern_init的入口地址，因为不存在函数堆栈的栈帧切换，那么在查看运行的堆栈结果时，自然就查看不到bootmain到kern_init的栈帧切换了。

练习6

Q：第121号中断，其处理程序的特权级为什么可以是ring 3？或者说中断处理程序的特权级都必须是ring 0吗?

A：只有系统调用的特权级是ring 3，这个特权级下只能使用int 0x30指令。在某个用户态程序触发中断时，CPU将对门（或者说门所在代码段）的DPL和程序的CPL进行额外检查，前者必须低于或等于后者，否则程序无法访问该中断的中断处理程序且将产生一个一般保护异常。

Q：如何用IDTR找到IDT？SETGATE宏当中gate参数的含义是什么?

A：从IDTR寄存器中可以直接得到中断处理程序所在代码段的基地址；

初始化时gate参数取为idt数组的一项，就是一个gatedesc结构体。

challenge1&2

Q：为什么init.c中switch_to_user要先把esp-8？而switch_to_kernel不用？或者，换个问法：在内核态切换到用户态需要修改ss，为什么用户态到内核态不用修改ss？

A：在中断返回时，先弹出eip和cs，再根据cs判断会返回到什么特权级。如果要返回到更低的特权级，会弹出ss和esp。用户态的cs=1b=00011011，CPL=DPL=RPL=3，内核态的cs=08=00001000，CPL=DPL=RPL=0。所以内核态到用户态的过程中，是从CPL=0切换到DPL=3，弹出esp和ss，这里弹出的ss是user_ss，存放在ss中，实现内核态到用户态的切换。但是在用户态到内核态的过程中，是从CPL=3切换到DPL=0，是返回到一个更高的特权级，所以本身没有esp和ss被弹出，又因为在用户态下中断，TSS会帮助把用户态的esp和ss压栈，并把当前的ss切换为内核态的ss。刚进入中断的时候我们的ss已经变成内核态的ss了，自然不用修改。

前置

在解压后的ucroe源码包中使用make命令即可以生成所需的目标文件,例如在本次实验中

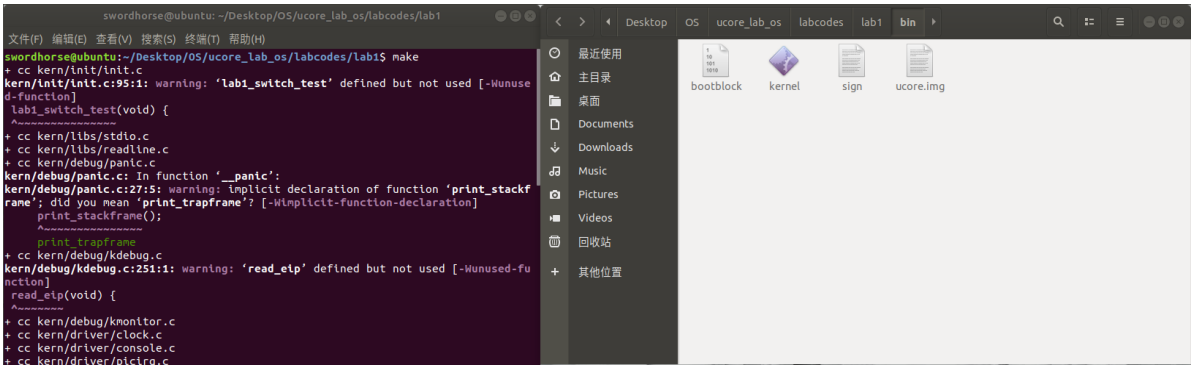
```
user@system:~../lab1$ make
```

之后就会生成一系列的目标文件：

- user.img : 被qemu访问的虚拟硬盘文件
- kernel : ELF格式的toy ucore kernel执行文，嵌入到了ucore.img中
- bootblock : 虚拟的硬盘主引导扇区(512字节)，包含了bootloader执行代码，同样嵌入了
- sign : 外部执行程序，用来生成虚拟的硬盘主引导扇区

还有其他文件，不一一列举。

如果要对修改后的ucore代码和ucore源码进行比较，可以使用diff命令。



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)
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。

操作系统镜像文件ucore.img是如何一步一步生成的?

通过使用以下命令，可以得到Makefile中具体执行的所有命令，之后就可以对每一条命令进行分析。

```
$ make "V="
```

通过这个命令会弹出来一长串信息，这个我们先不看，还是先从makefile文件入手。打开makefile文件，一下子看到一堆代码也是挺烦人的，不过可以发现里面写了注释。既然有注释，就好办多了，我们知道通过这个命令可以生成一个ucore.img文件，那我们就从ucore.img倒推回去，阅读注释，可以看到以下代码。

```
# create ucore.img
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)
```

即使我们不懂makefile的语法规则，我们也很容易知道要生成这个ucore.img需要kernel和bootblock两个文件，那就再分别从这两个文件往上追溯。

我们先从bootblock来看，bootblock需要生成bootasm.o,bootmain.o以及sign等文件

```
# create bootblock
bootfiles = $(call listf_cc,boot)
$(foreach f,$(bootfiles),$(call cc_compile,$(f),$(CC),$(CFLAGS) -Os -nostdinc))

bootblock = $(call totarget,bootblock)

$(bootblock): $(call toobj,$(bootfiles)) | $(call totarget,sign)
        @echo + ld $@
        $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ -o $(call toobj,bootbl
ock)
        @$(OBJDUMP) -S $(call objfile,bootblock) > $(call asmfile,bootblock)
        @$(OBJCOPY) -S -O binary $(call objfile,bootblock) $(call outfile,bootbl
ock)
        @$(call totarget,sign) $(call outfile,bootblock) $(bootblock)

$(call create_target,bootblock)
```

其中bootasm.o由bootasm.S生成，bootmain.o由bootmain.c生成，生成代码为

```
gcc -Iboot/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc
-fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootasm.S -o
obj/boot/bootasm.o

gcc -Iboot/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc
-fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootmain.c -o
obj/boot/bootmain.o
```

解释一下其中出现的参数

```
-fno-builtin     # 不承认不以__builtin_开头的函数为内建函数
-fno-PIC     # 产生与位置无关代码，即没有绝对地址，使代码可以被任意加载
-Wall    # 在编译后显示所有警告
-ggdb    # 生成专门用于gdb的调试信息
-m32     # 生成32位机器的汇编代码
-gstabs # 以stabs格式生成调试信息
-nostdinc    # 不在标准系统文件夹中寻找头文件，只在-I中指定的文件夹搜索头文件
-I   # 添加搜索头文件的路径并且会被优先查找
-Os # 优化代码，减小大小
-c   # 把程序做成obj文件，就是.o
-o   # 制定目标名称
-fno-stack-protector     # 不生成用于检测缓冲区溢出的代码
```

生成sign的代码如下，编写在makefile文件中

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

对应的命令是，因为没有新的参数，就不进行详细解释。

```
gcc -Itools/ -g -Wall -O2 -c tools/sign.c -o obj/sign/tools/sign.o
gcc -g -Wall -O2 obj/sign/tools/sign.o -o bin/sign
```

整个的bootblock的生成过程如下

```
# 生成bootblock.o
# 新参数 -m:模拟为i386上的链接器，-N:设置代码段和数据段可读可写，-e:指定入口，-Ttext:设置代码开始位置
ld -m    elf_i386 -nostdlib -N -e start -Ttext 0x7C00 obj/boot/bootasm.o obj/boot/bootmain.o -o obj/bootblock.o

# 将bootblock.o拷贝到bootblock.out
# 新参数 -S:移除所有符号和重定位信息，-O:指定输出格式
objcopy -S -O binary obj/bootblock.o obj/bootblock.out

# 使用sign处理bootblock.out生成bootblock
bin/sign obj/bootblock bin/bootblock
```

再看kernel的相关代码如下：

```
# kernel

KINCLUDE        += kern/debug/ \
                   kern/driver/ \
                   kern/trap/ \
                   kern/mm/

KSRCDIR         += kern/init \
                   kern/libs \
                   kern/debug \
                   kern/driver \
                   kern/trap \
                   kern/mm

KCFLAGS         += $(addprefix -I,$(KINCLUDE))

$(call add_files_cc,$(call listf_cc,$(KSRCDIR)),kernel,$(KCFLAGS))

KOBJS   = $(call read_packet,kernel libs)

# create kernel target
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)
```

注意到KSRCDIR这一部分的内容实际上是用给定目录的方式进行对.c文件的添加，在被执行的时候就会在这些目录中选择没使用过的.c文件来编译成.o文件。之后kernel对这些所有的.o文件进行一个链接。

生成完kernel和bootblock之后，就该生成ucore.img了。由上面的生成代码即

```
# 生成一个有10000块的ucore.img文件，每个块默认大小为512字节
dd if=/dev/zero of=bin/ucore.img count=10000
# 把bootblock添加到ucore.img的第一个块之中
dd if=bin/bootblock of=bin/ucore.img conv=notrunc
# 把kernel写到ucore.img的其它块中
dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
# 其中几个关键参数的意义
if:输入文件，不指定从stdin中读取
of:输出文件，不指定从stdout中读取
/dev/zero:不断返回的0值
count:块数
conv = notrunc:输出不截断
seek = num:从输出文件开头跳过num个块
```

这样我们就知道了整个ucore.img是如何从无到有的。

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?

主引导扇区就是我们的bootblock被加载到的区域，而和生成bootblock有关的代码就是sign.c。查看这个文件得到如下代码

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/stat.h>
int main(int argc, char *argv[]) {
    struct stat st;
    if (argc != 3) {
        fprintf(stderr, "Usage: <input filename> <output filename>\n");
        return -1;
    }
    if (stat(argv[1], &st) != 0) {
```

```
        fprintf(stderr, "Error opening file '%s': %s\n", argv[1],
strerror(errno));
        return -1;
    }
    printf("'%s' size: %lld bytes\n", argv[1], (long long)st.st_size);
    if (st.st_size > 510) {
        fprintf(stderr, "%lld >> 510!!\n", (long long)st.st_size);
        return -1;
    }
    char buf[512];
    memset(buf, 0, sizeof(buf));
    FILE *ifp = fopen(argv[1], "rb");
    int size = fread(buf, 1, st.st_size, ifp);
    if (size != st.st_size) {
        fprintf(stderr, "read '%s' error, size is %d.\n", argv[1], size);
        return -1;
    }
    fclose(ifp);
    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;
    }
    fclose(ofp);
    printf("build 512 bytes boot sector: '%s' success!\n", argv[2]);
    return 0;
}
```

通过分析上面这段代码，我们可以得到一个合格的主引导扇区应该符合如下两个规则：

- 输入字节在510字节内
- 最后两个字节是0x55AA

练习2：使用qemu执行并调试lab1中的软件(简要写出练习过程)

为了熟悉使用qemu和gdb进行的调试工作，我们进行如下的小练习：

1. 从CPU加电后执行的第一条命令开始，单步跟踪BIOS的执行
2. 在初始化位置0x7c00设置实地址断点，测试断点正常
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
4. 自己找一个booloader或内核中的代码位置，设置断点并进行测试。

解题过程：

1. **启动gdb，连接到qemu进行远程调试**

   gbd调试

   摘录部分常用命令和参数如下：

   - list <linenum>，显示程序第linenum行周围的源程序；list <function>，显示函数名为function的函数的源程序；list，显示当前行后面的源程序；list -，显示当前行前面的源程序
   - path <dir>，设定程序运行路径；how paths查看路径

- cd <dir>，相当于shell的cd命令；pwd显示当前所在目录
- break [filename:]<function>或[filename:]<linenum>，在源文件（可选参数）的某个函数或某行停住；break +offset或-offset，在当前行号的前面或后面的offset行停住，offset为自然数；break *address，在程序运行的内存地址停住；break，无参数时表示在下一条指令处停住；break ... if <condition>，以上命令均可与if语句配合使用，使得满足一定条件时在指定位置停住程序
- info break[n]或breakpoints[n]，查看第n个断点；info break，列出当前所设置的所有观察点
- 单步调试：next，相当于VC++当中的step over；step，相当于step into
- continue或c或fg：继续执行程序直到程序结束或到达下一个断点
- x /nfu [addr]，显示指定地址addr及其附近的内容，其中n表示机器指令（汇编码）个数，f表示格式（包括十六进制x、字符串s、指令i等），u表示单元大小（b：1B，h：2B，w：4B，g：8B），如果不显式指定addr，则地址默认为上一次x命令显示之后的地址。
- layout，打开可视化窗口；layout asm，打开反汇编窗口；ctrl+x a，退出当前可视化窗口回到终端

所用gdb命令：

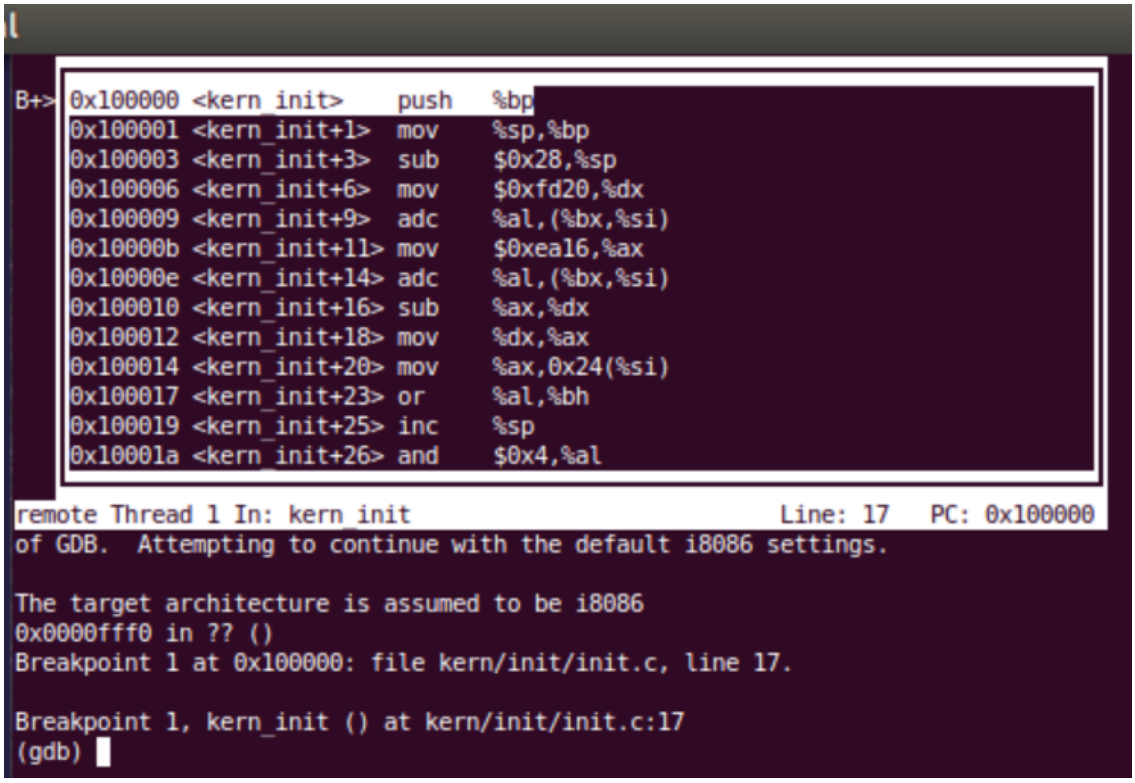
```
file bin/kernel #指定调试目标文件，让gdb获得符号信息
target remote :1234 #设置远程连接端口为qemu的运行端口1234，连接到qemu
set architecture i8086 #指定qemu要模拟的硬件架构
b *0x7c00 #在bootloader开始地址0x7c00处下断点
continue #开始调试，执行到刚才指定的断点
x /2i $pc #以十六进制格式打印当前机器指令及其下方一条机器指令的地址，并显示汇编

layout asm #显示汇编可视化窗口
```

2. **BIOS启动：gdb查看启动后第一条执行的指令并查看BIOS代码**

   修改gdbinit中指令为，在lab1目录下执行make debug命令启动qemu，程序在启动后第一条指令停住。（*按照原来文件的命令，停在了0x100000，kern_init函数的入口地址，应该不符合题目要求，题目答案改成了break bootmain*）

   kernel 加载进来时用.ld文件链接操作系统

   bootmain函数的功能?



查看后续BIOS代码：

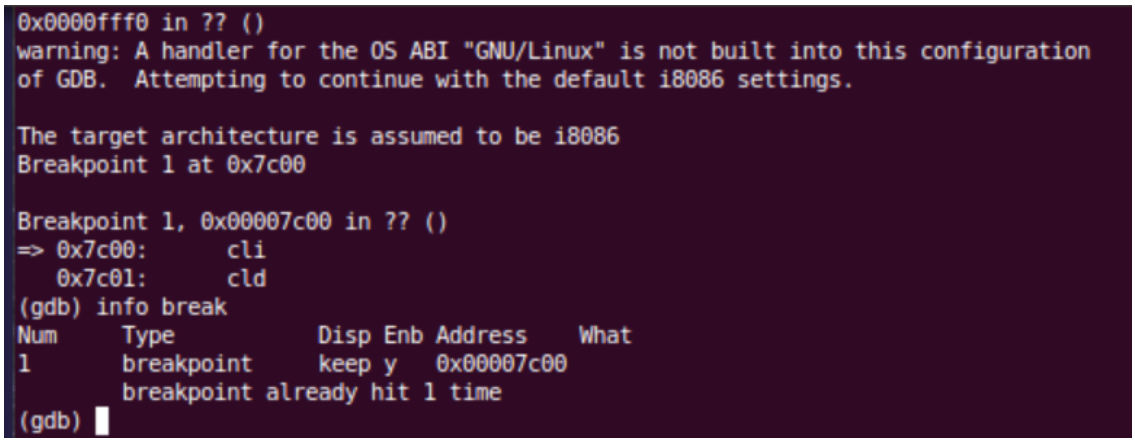
执行类似x /10i addr的命令即可

```
file obj/bootblock.o
target remote:1234
break bootmain
continue
```

3. **跳转到bootloader：在0x7c00处设置断点、测试正常可用**

   如图。执行make lab1-mon命令后可在0x7c00处停住，并能按照指定规则打印出相应的汇编码。

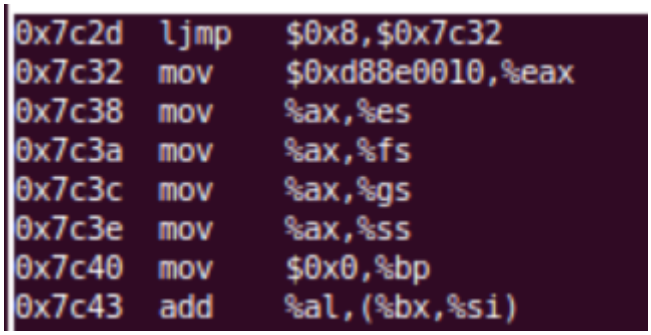
   这个地址在lab1init文件中显式给出。



4. **单步调试+反汇编，跟踪代码运行，将调试时得到的反汇编代码与bootasm.S和bootblock.asm进行比较。**

   bootasm.s中包括的定义：内核代码段选择子、内核数据段选择子、保护模式使能标志、全局描述符表

bootasm.s中包括的功能代码或代码块：禁止中断，设置寻址方向为朝向高地址，初始化（清空）DS, ES, SS段寄存器，A20使能，保护模式下初始化（设为保护模式的数据段选择子）数据段寄存器DS, ES, FS, GS, SS，初始化一个栈的指针并调用bootmain.c中的bootmain函数执行bootloader（如果这个函数意外地返回则在下方汇编代码里进入死循环）



可视化窗口的汇编风格是x86的，而bootasm.s文件是AT&T写法。

——如上图，可视化窗口来到保护模式下初始化的时候，没看到DS的初始化，但s文件里是有的

```
0x7c4a  call 0x7cfe #bootmain函数的起始地址应该在此处，但这个地址上的汇编是pop %bp？？
```

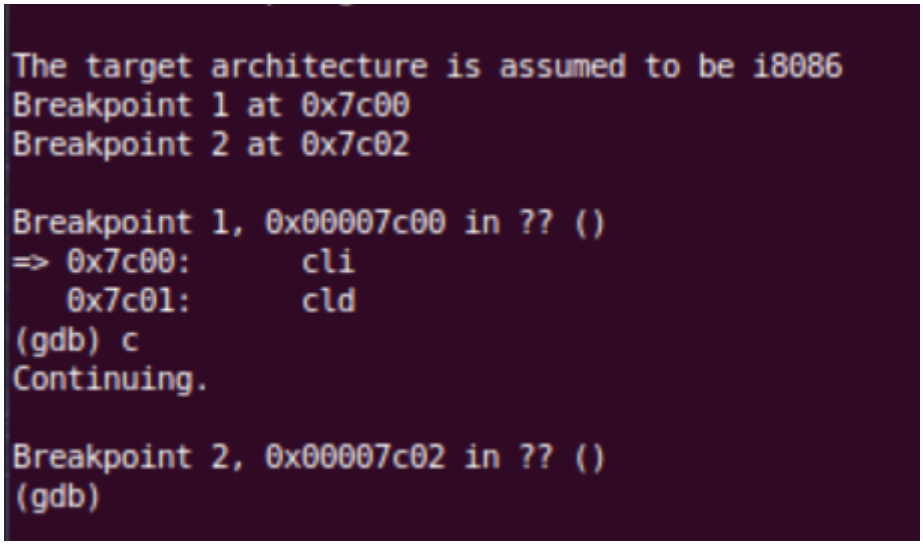
```
0x7c4f  jmp 0x7c4f #接下来是一个死循环
```

编译lab1中的代码，在其中obj文件夹下找到**bootblock.asm**，即bootloader的汇编代码源文件。

看到各指令下方均标明了所在地址和对应的十六进制机器码，和可视化窗口中的能够相互对应。

这个文件的105行声称bootmain的起始地址是7cd1？？

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



仿照之前的断点命令格式，在lab1init文件中添加一个断点，地址是0x7c02，测试可用。

练习3：分析bootloader进入保护模式的过程(写出分析)

BIOS将通过读取硬盘主引导扇区到内存，并跳转到对应内存中的执行位置执行bootloader。请分析bootloader是如何完成从实模式进入保护模式的。

既然题目中都给了提示要看bootasm.S的代码，那我们就先从这个源码入手，代码内容如下：

```
#include <asm.h>

# Start the CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.

.set PROT_MODE_CSEG,        0x8                     # kernel code segment
selector
.set PROT_MODE_DSEG,        0x10                    # kernel data segment
selector
.set CR0_PE_ON,             0x1                     # protected mode enable flag

# start address should be 0:7c00, in real mode, the beginning address of the
running bootloader
.globl start
start:
.code16                                             # Assemble for 16-bit mode
    cli                                             # Disable interrupts
    cld                                             # String operations
increment

    # Set up the important data segment registers (DS, ES, SS).
    xorw %ax, %ax                                   # Segment number zero
    movw %ax, %ds                                   # -> Data Segment
    movw %ax, %es                                   # -> Extra Segment
    movw %ax, %ss                                   # -> Stack Segment

    #  Enable A20:
    #  For backwards compatibility with the earliest PCs, physical
    #  address line 20 is tied low, so that addresses higher than
    #  1MB wrap around to zero by default. This code undoes this.
seta20.1:
    inb $0x64, %al                                  # Wait for not busy(8042
input buffer empty).
    testb $0x2, %al
    jnz seta20.1

    movb $0xd1, %al                                 # 0xd1 -> port 0x64
    outb %al, $0x64                                 # 0xd1 means: write data to
8042's P2 port

seta20.2:
    inb $0x64, %al                                  # Wait for not busy(8042
input buffer empty).
    testb $0x2, %al
    jnz seta20.2

    movb $0xdf, %al                                 # 0xdf -> port 0x60
    outb %al, $0x60                                 # 0xdf = 11011111, means set
P2's A20 bit(the 1 bit) to 1

    # Switch from real to protected mode, using a bootstrap GDT
    # and segment translation that makes virtual addresses
    # identical to physical addresses, so that the
    # effective memory map does not change during the switch.
    lgdt gdtdesc
```

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

    # Jump to next instruction, but in 32-bit code segment.
    # Switches processor into 32-bit mode.
    ljmp $PROT_MODE_CSEG, $protcseg

.code32                                             # Assemble for 32-bit mode
protcseg:
    # Set up the protected-mode data segment registers
    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

    # Set up the stack pointer and call into C. The stack region is from 0--
start(0x7c00)
    movl $0x0, %ebp
    movl $start, %esp
    call bootmain

    # If bootmain returns (it shouldn't), loop.
spin:
    jmp spin

# Bootstrap GDT
.p2align 2                                          # force 4 byte alignment
gdt:
    SEG_NULLASM                                     # null seg
    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:
    .word 0x17                                      # sizeof(gdt) - 1
    .long gdt                                       # address gdt
```

这段代码是bootmain执行之前bootloader所做的工作。不过在正式读代码之前，先读注释。从注释之中我们可以看到我们的代码分别完成了以下几个部分的功能：

```
# 第一部分:屏蔽中断，设置串地址增长方向，设置一些重要的数据段寄存器(DS,ES,SS)
# start address should be 0:7c00, in real mode, the beginning address of the
running bootloader
.globl start
start:
.code16                                             # 使用16位模式编译
    cli                                             # 屏蔽中断
    cld                                             # 设置串地址增长方向

    xorw %ax, %ax                                   # Segment number zero
    movw %ax, %ds                                   # -> Data Segment
    movw %ax, %es                                   # -> Extra Segment
    movw %ax, %ss                                   # -> Stack Segment
```

在第一部分中处于实模式下，可用的内存大小不多于1M，因此需要告诉编译器使用16位模式编译。cli是禁用中断。cld将DF位置零，从而决定内存地址是增大(对应的std是将DF置一，内存地址减小)。之后使用xorw异或指令让ax寄存器值变成0，再把ax的值赋给ds，es和ss寄存器。准备工作到此结束

```
 # 第二部分:启动A20
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
```

首先要说一下A20是一个什么东西:

- 最早的8086结构中的内存空间很小，一开始8086的地址线有20位，也就是说具有0-1M的寻址范围，不过当时的寄存器只有16位，无法满足寻址需求，所以采用了另外一种寻址方式：一个16位寄存器表示基址*16+另外一个16位寄存器表示偏移地址，这样寻址空间就超过了1M。但到了后来，地址线增加到了32位，为了让以前的机器也能使用这种方式(即向下兼容)，就在A20(第20根地址线)上做了一个开关，当A20被使能时，是一根正常的地址线，但是当不被使能时永远为零。在保护模式下，要访问高端的内存必须要打开这个开关，否则第21位一定是0.
- 8086体系结构的地址空间实际上是被“挖洞”了的。最早的1M内存被分为了640KB的低端常规内存和384KB的留给ROM和系统设备的高端内存，然后这个不具有前瞻性的设计就导致在之后的内存容量增大非常麻烦:被划分成了0-640KB,1M-最大内存的两个部分。为了解决这个问题，采用了这样的解决办法:加电之后先让ROM有效，取出ROM之后再让RAM有效，把这部分内容保存到RAM这部分地址空间中。
- 实际上A20是由一个8042键盘控制器来控制的A20 Gate，8042芯片有三个端口，其中之一是Output Port，而A20Gate就是Output Port端口的 bit1，所以控制A20的方式就是通过读写端口数据，使bit1为1。

再讲讲这个8042芯片。

- 这个芯片有两个外部端口0x60h和0x64h，相当于读写操作的地址。
- 在读Output Port时，需要先向0x64h发送0d0h命令，然后从0x60h读取Output Port的内容；
- 在写Output Port时，需要先向0x64h发送0d1h命令，然后往0x60h写入Output Port的内容。
- 在读写的同时还需要检查缓冲区是否有数据，有的话就暂停等待。

有了上面的内容我们看这部分代码就比较简单了，每一步的作用写在注释中了。

```
# 第二部分:启动A20
seta20.1:
    inb $0x64, %al          #读取当前状态到al寄存器
    testb $0x2, %al         #检查当前状态寄存器的第二位是否为1(缓冲区是否为空)
    jnz seta20.1            #若缓冲区不为0，跳转到开始处
    movb $0xd1, %al         #将0xd1h写入al寄存器
    outb %al, $0x64         #向0x64h发送0xd1h命令，表示要写
seta20.2:
    inb $0x64, %al          #同1
    testb $0x2, %al         #同1
    jnz seta20.2            #同1
    movb $0xdf, %al         #将0xdfh写入al寄存器
    outb %al, $0x60         #向0x60h写入0xdfh，打开A20
```

之后就是第三部分，初始化GDT表，通过lgdt gdtdesc指令就可以实现。

接下来第四部分就是进入保护模式，进入保护模式的原理就是让cr0寄存器中的PE值为1

```
 # 第四部分
 movl %cr0, %eax
 orl  $CR0_PE_ON, %eax
 movl %eax, %cr0
```

之后的第五部分通过一个长跳转来更新CS寄存器的基地址

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

我们可以注意到代码段最前面定义了PROT_MODE_CSGE和PROT_MODE_DSEG，分别被定义为0x8h和0x10h，这两个分别是代码段和数据段的选择子。

第六部分是设置段寄存器并建立堆栈

```
# 第六部分:初始化各个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
# 第七部分:调用bootmain
    call bootmain
spin:
    jmp spin
```

这个就是将各个段寄存器设置为0x10h，ebp指向0x0h，esp指向start也就是0x7c00处，最后使用call函数将返回地址入栈，控制权交给bootmain。

最后一个spin是如果当bootmain异常返回，在这里循环。

练习4：分析bootloader加载ELF格式的OS过程(写出分析)

通过阅读bootmain.c,了解bootloader如何加载ELF文件，通过分析源代码和通过qemu来运行并调试bootloader&OS

- bootloader是如何读取硬盘扇区?
- bootloader是如何加载ELF格式的OS?

理论部分

kernel是一个elf文件，因此需要理解bootloader是如何从磁盘扇区读取kernel并在读取后进行分析的。

```
/* readsect - read a single sector at @secno into @dst */
static void
readsect(void *dst, uint32_t secno) {
    // wait for disk to be ready 等磁盘准备好
    waitdisk();
//把参数设置好，明确读取磁盘的命令
outb(0x1F2, 1);                         // count = 1
outb(0x1F3, secno & 0xFF);
outb(0x1F4, (secno >> 8) & 0xFF);
outb(0x1F5, (secno >> 16) & 0xFF);
outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
outb(0x1F7, 0x20);                      // cmd 0x20 - read sectors

// wait for disk to be ready
waitdisk();

// read a sector 如果0x1F7不忙的话就从0x1F0把磁盘扇区数据读取到相应的内存上去
insl(0x1F0, dst, SECTSIZE / 4);
```

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

bootloader通过readsect函数来读取磁盘扇区，用到了内联汇编的in和out系列函数。所有的IO操作是通过CPU访问硬盘的IO地址寄存器完成，其中访问第一个硬盘的扇区是通过设置IO地址寄存器0x1f0-0x1f7实现的，每个通道的主从盘的选择通过第6 个IO偏移地址寄存器来设置，地址的第6位如果是1，那就是LBA模式，为0就是CHS模式；而readsec函数中用到的in和out函数的参数表也如下所示。

/* insl:从I/O端口port读取count个数据(单位双字)到以内存地址addr为开始的内存空间

*/ void insl(unsigned port, void *addr, unsigned long count);

/* outb:写字节端口(8位宽)。

*/ void outb(unsigned char byte, unsigned port);

```
//readseg函数实现了从offset地址处读取count个字节的数据到虚拟地址va处的功能
static void
readseg(uintptr_t va, uint32_t count, uint32_t offset) {
    uintptr_t end_va = va + count;
// round down to sector boundary
va -= offset % SECTSIZE;

// translate from bytes to sectors; kernel starts at sector 1
//因为sector1从1开始，所以+1不能忘
uint32_t secno = (offset / SECTSIZE) + 1;

// If this is too slow, we could read lots of sectors at a time.
// We'd write more to memory than asked, but it doesn't matter --
// we load in increasing order.
for (; va < end_va; va += SECTSIZE, secno ++) {
    readsect((void *)va, secno);}
}
```

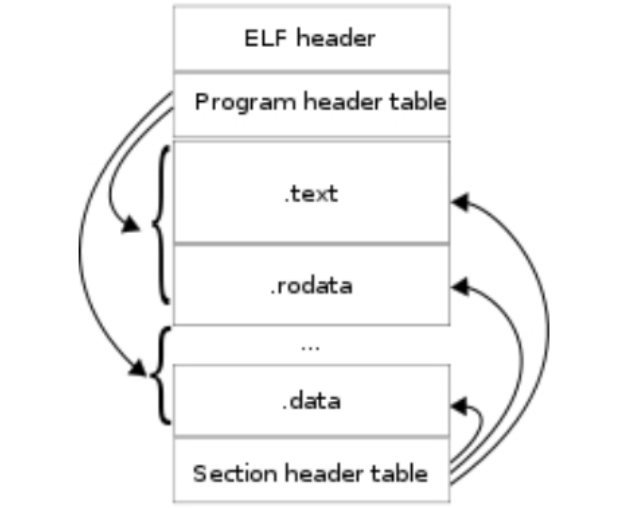
readseg读入好多个字节，并转化成扇区可以读的大小，把值传给readsect代码。最终的读取扇区工作由readsect实现。

```
/* bootmain - the entry of bootloader */
void
bootmain(void) {
    // read the 1st page off disk
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);
// is this a valid ELF?
if (ELFHDR->e_magic != ELF_MAGIC) {
    //判断e_magic是不是ELF_MAGIC类型，如果不是的话说明文件无效
    goto bad;
}

struct proghdr *ph, *eph;

// load each program segment (ignores ph flags)
ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
//e_phoff是program header表的偏移位置，所以现在ph找到了program header表的实际位置
eph = ph + ELFHDR->e_phnum;  //e_phnum是表中的入口数目
for (; ph < eph; ph ++) {
    //p_va是段的第一个字节将被放到内存中的虚拟地址；
    //p_memsz是段在内存映像中占用的字节数；
    //p_offset是段相对文件头的偏移值；
    //readseg(uintptr_t va, uint32_t count, uint32_t offset)对照段的读取函数进行参数输入
    readseg(ph->p_va & 0xFFFFFF, ph->p_memsz, ph->p_offset);
}

// call the entry point from the ELF header
// note: does not return
//运行程序入口的虚拟地址
((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();
bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);
    /* do nothing */
 while (1);
}
```

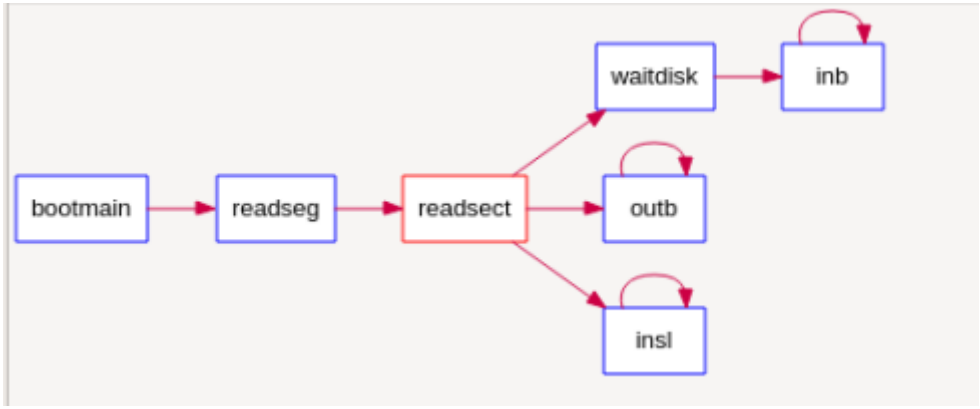
由上述代码可以得出bootloader加载ELF格式文件的步骤：

1. 先判断是不是有效的ELF文件；
2. 通过elf的文件头找到program header表；
3. 根据program header表中的每个段的内存映像地址、大小等，并读取段数据（如数据段、代码段等）
4. 必要的数据读取完成后跳转到程序入口的虚拟地址准备运行。

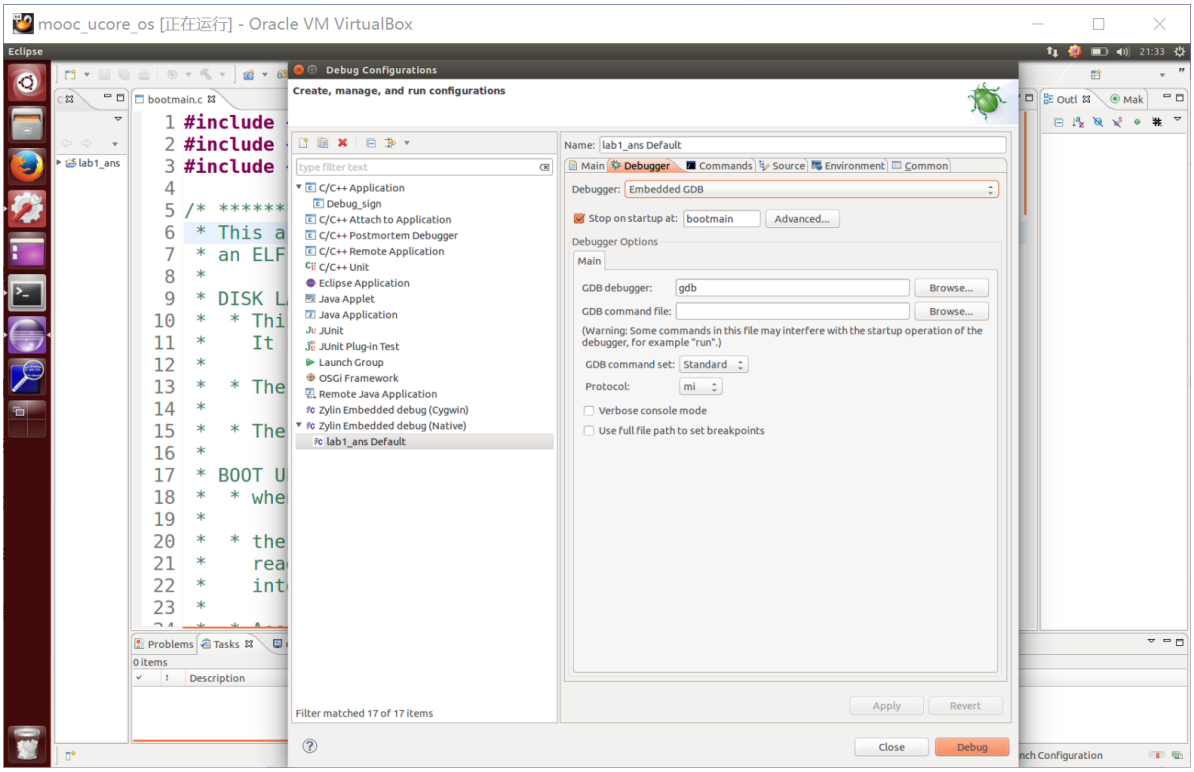
从ELF文件格式可以更清晰地看到bootloader在判断完elf文件类型后，跳转到相应的段进行后需磁盘访问。



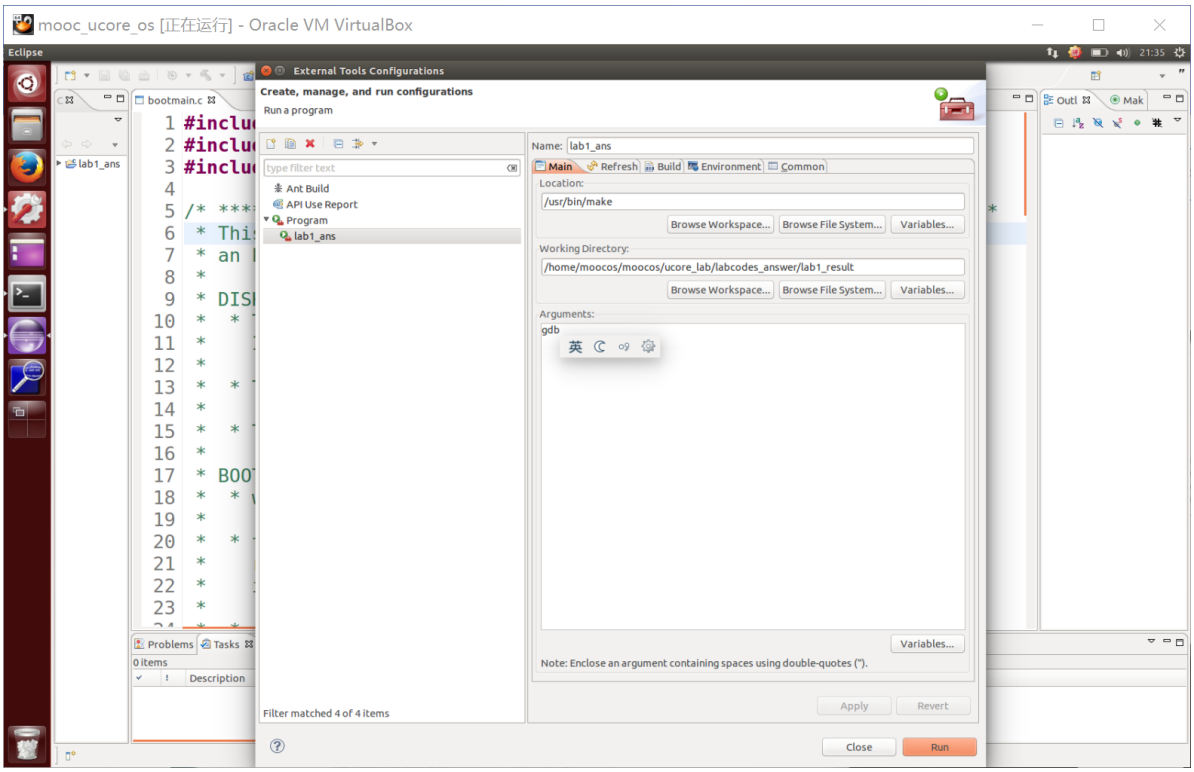
我们还可以利用understand可以查看regsect的函数调用



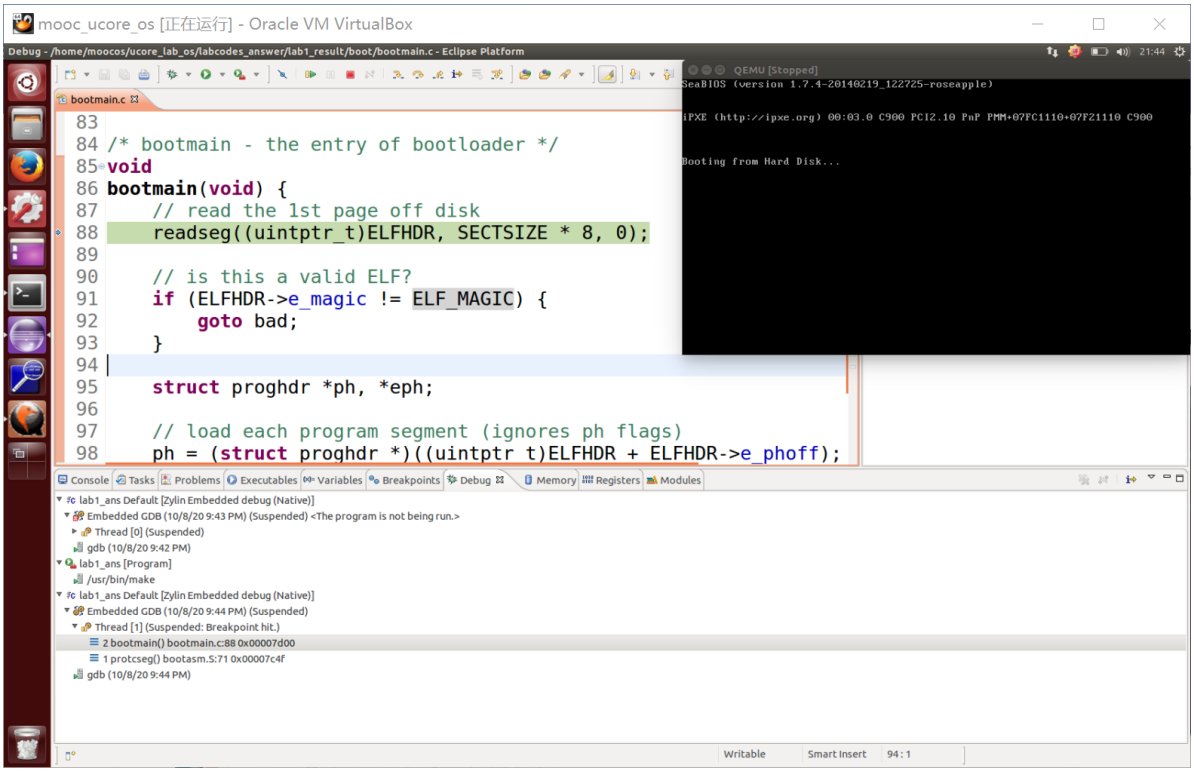
实践调试部分



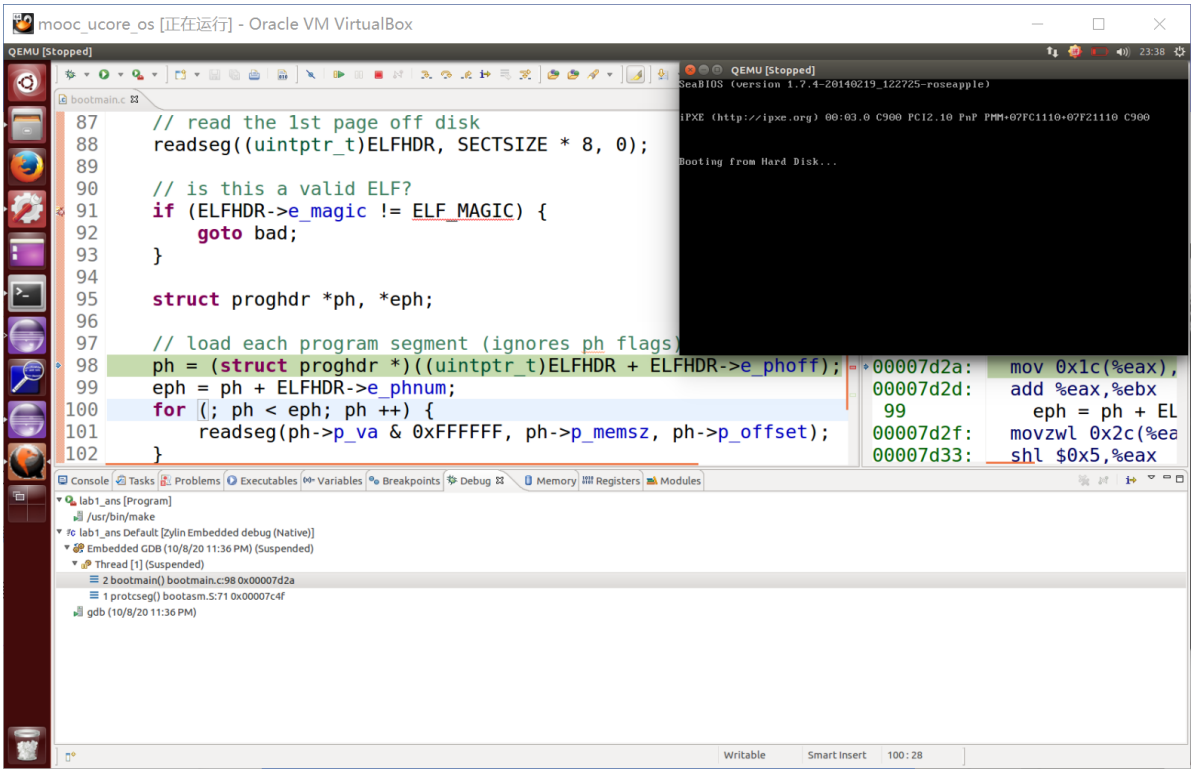
运用可视化窗口进行bootloader的debugger



使用make工具打开qemu



打开qemu准备好启动，并开启debugger，可以看到程序现在依照之前设定好的停在了bootmain里面。



step over调试，此时ph=0x0，

```
98      ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
99      eph = ph + ELFHDR->e_phnum;
```

往下一步，可得ph=0x10034，ELFHDR->e_phnum=3

eph=0x10094

```
for (; ph < eph; ph ++) {
    readseg(ph->p_va & 0xFFFFFF, ph->p_memsz, ph->p_offset);
}
```

在循环中，ph->p_va=0x0，ph->p_memsz=0，ph->p_offset=0

```
((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();
```

循环结束后，查看得ELFHDR->e_entry=1048576

练习5：实现函数调用堆栈跟踪函数(需要编程)

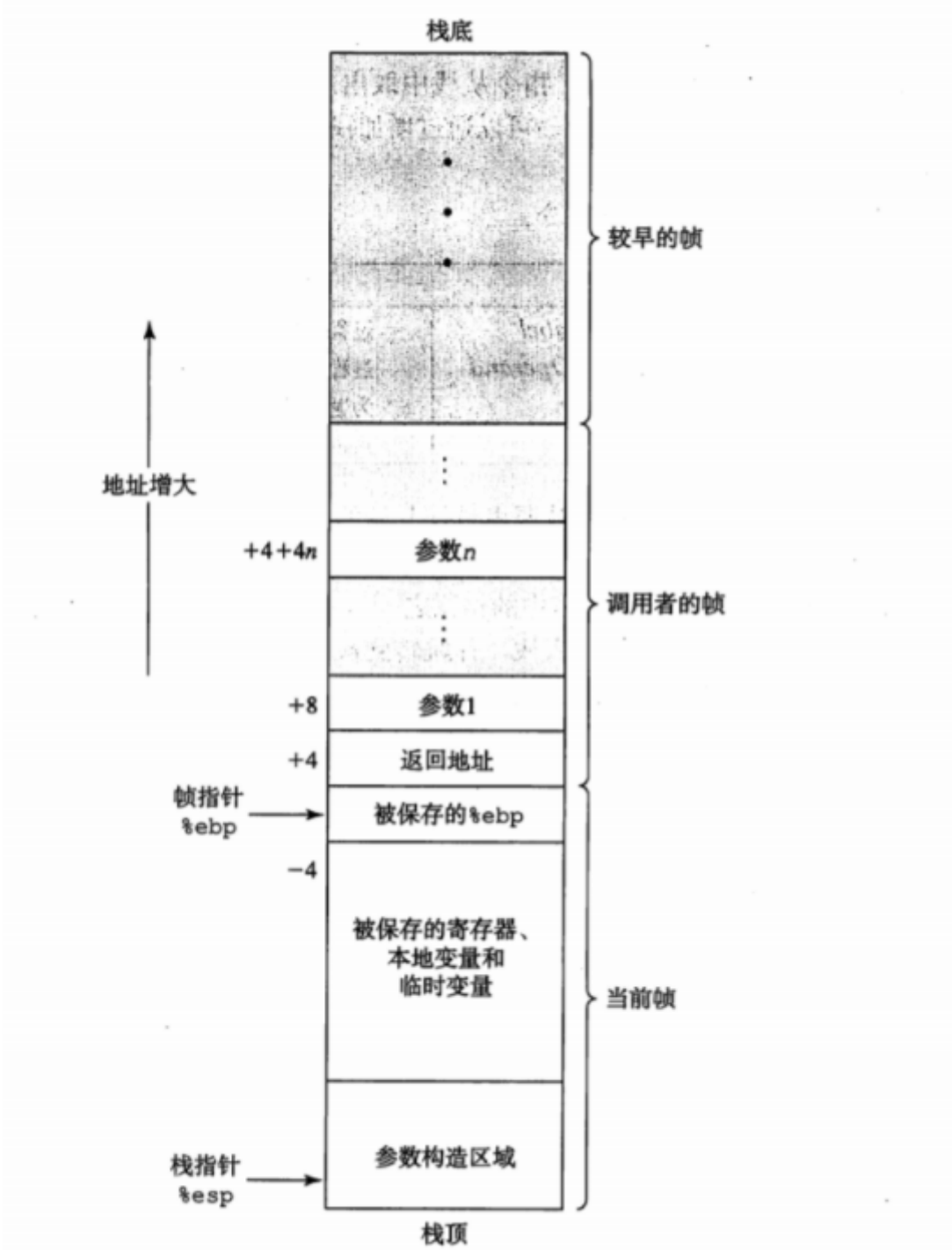
我们需要在lab1中完成kdebug.c中函数print_stackframe的实现，可以通过函数print_stackframe了跟踪函数调用堆栈中记录的返回地址。

在lab1/kern/debug目录下找到kdebug.c,打开以后发现源文件中已经有一个print_stackframe函数了(虽然里面啥也没有)，要做的就是往里面添加代码，让这个函数能实现我们需要的功能。以下是初始状态的代码。

```
void print_stackframe(void) {
     /* 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
(uint32_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]
      */
}
```

为了实现函数调用堆栈跟踪函数，需要先了解函数调用栈的原理。

函数调用时，自栈顶(低地址)到栈底(高地址)的情况如下图所示



栈帧结构（栈用来传递参数、存储返回信息、保存寄存器，以及本地存储）

esp和ebp两个指针是其中最关键的部分，只要掌握了ebp和esp的位置，就能很容易理解函数调用过程了。根据这个图，我们可以有这样几个信息

- ss[ebp]指向上一层的ebp
- ss[ebp-4]指向局部变量
- ss[ebp+4]指向返回地址
- ss[ebp+4+4n]指向第n个参数

之后我们就可以着手实现堆栈跟踪函数了。首先我们知道在bootasm.S中将esp设置为0x7c00，ebp设置为0，就调用了bootmain函数。call指令会依次执行以下命令:push返回地址，push这一层的ebp，然后把现在的esp赋值给ebp。在执行完call之后，这个ebp指向了0x7bf8(0x7c00-4-4)。

实现之前我们看一下源代码中的注释，突然惊讶的发现注释已经非常贴心的教你怎么写了，那就按着这个注释一步一步来，写出如下的结果：

```
void
print_stackframe(void)
{
     /* 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
(uint32_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]
      */
    uint32_t ebp = read_ebp(), eip = read_eip();
    for(int 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;
        for(int j = 0; j < 4; j++){
            cprintf("0x%08x ", args[j]);
        }
        cprintf("\n");
        print_debuginfo(eip - 1);
        eip = ((uint32_t *)ebp)[1];
        ebp = ((uint32_t *)ebp)[0];
    }
}
```

之后在lab1目录下执行命令 $ make qemu，得到如下的输出

```
......
Kernel executable memory footprint: 64KB
ebp=: 0x00007b28 | eip=: 0x00100a63 | args=: 0x00010094 0x00010094 0x00007b58
0x00100092
    kern/debug/kdebug.c:307: print_stackframe+21
ebp=: 0x00007b38 | eip=: 0x00100d4d | args=: 0x00000000 0x00000000 0x00000000
0x00007ba8
    kern/debug/kmonitor.c:125: mon_backtrace+10
ebp=: 0x00007b58 | eip=: 0x00100092 | args=: 0x00000000 0x00007b80 0xffff0000
0x00007b84
    kern/init/init.c:48: grade_backtrace2+33
ebp=: 0x00007b78 | eip=: 0x001000bc | args=: 0x00000000 0xffff0000 0x00007ba4
0x00000029
    kern/init/init.c:53: grade_backtrace1+38
ebp=: 0x00007b98 | eip=: 0x001000db | args=: 0x00000000 0x00100000 0xffff0000
0x0000001d
    kern/init/init.c:58: grade_backtrace0+23
ebp=: 0x00007bb8 | eip=: 0x00100101 | args=: 0x001032dc 0x001032c0 0x0000130a
0x00000000
    kern/init/init.c:63: grade_backtrace+34
ebp=: 0x00007be8 | eip=: 0x00100055 | args=: 0x00000000 0x00000000 0x00000000
0x00007c4f
    kern/init/init.c:28: kern_init+84
```

```
ebp=: 0x00007bf8 | eip=: 0x00007d72 | args=: 0xc031fcfa 0xc08ed88e 0x64e4d08e
0xfa7502a8
    <unknow>: -- 0x00007d71 --
......
```

最后一行中给出了ebp，eip和args三个参数，其具体意义为

- **ebp=: 0x00007bf8** 是跳转到bootmain
- **eip=: 0x00007d72** 是从bootasm.s跳转到bootmain前的地址，也就是bootmain的返回地址。
- **args=: 0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8** 通常状态下，args存放的四个dword是对应4个输入参数的值。但是再最底层处，即7c00往后增加的地址处，那里是bootloader的代码段，所以最后的args其实是bootloader指令的前十六个字节，下面这个例子就能很好的说明情况

```
# bootloader前三条指令对应的机器码
7c00:   cli             fa
7c01:   cld             fc
7c02:   xor    %eax,%eax    31 c0
# 由于是小端字节序，所以存储为 c0 31 fc fa
```

练习6：完善中断初始化和处理(需要编程)

请完成编码工作和回答如下问题：

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？

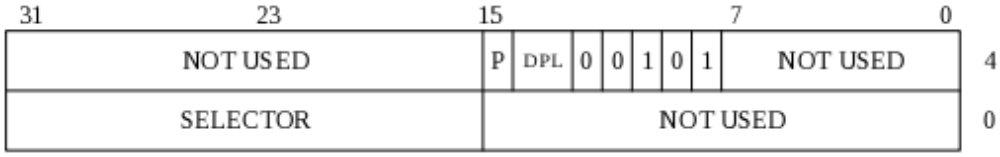
- CPU收到中断信息后，需要先根据中断类型码找到对应的中断处理程序地址，这个过程通过查中断向量表完成：该表的各项被称作中断向量，一个中断向量就是一个中断处理程序的起始地址，包括段基址（存储在高位）和偏移量（存储在低位），因此一个表项占**八个字节**，前四字节表示偏移量，后四字节表示段基址（段选择子）。
- 中断处理代码整体保存在一个段上，因此**第16-31位**段基址表示中断处理代码的入口。

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轰，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。
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“。

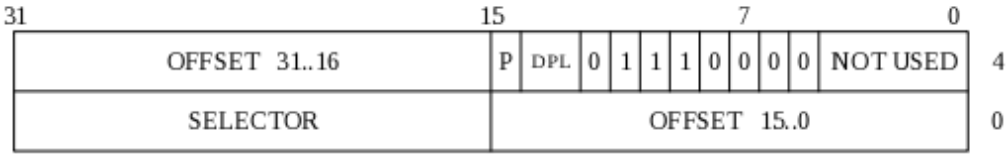
解答过程：

1. 完成编码工作并回答如下问题：
    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？**
        - 保护模式下，段寄存器含有段选择子；CPU收到中断信息后，需要先根据中断类型码找到对应的中断处理程序地址，这个过程通过查中断描述符表（获得特定中断处理程序的偏移量并以中断向量为索引查找中断处理程序的段选择子）以及全局描述符表（通过中断处理程序的段选择子获得段基址）完成；一个表项被称作一个门描述符，占**八个字节**。三种类型（还有第四种：调用门描述符，结构与任务门描述符相同）的门描述符结构如下：

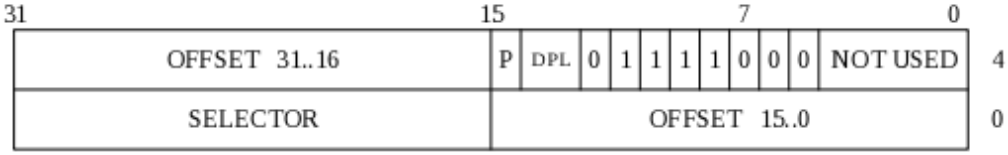
80386 TASK GATE



80386 INTERRUPT GATE



80386 TRAP GATE



- **第16-31位**（即低四字节中的高16位）段选择子结合**第0-15位、第48-63位**的偏移量可以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。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。

```
49  /* Gate descriptors for interrupts and traps */
50  struct gatedesc {
51      unsigned gd_off_15_0 : 16;        // low 16 bits of offset in segment
52      unsigned gd_ss : 16;             // segment selector
53      unsigned gd_args : 5;             // # args, 0 for interrupt/trap gates
54      unsigned gd_rsv1 : 3;             // reserved(should be zero I guess)
55      unsigned gd_type : 4;             // type(STS_{TG,IG32,TG32})
56      unsigned gd_s : 1;                // must be 0 (system)
57      unsigned gd_dpl : 2;             // descriptor(meaning new) privilege level
58      unsigned gd_p : 1;                // Present
59      unsigned gd_off_31_16 : 16;       // high bits of offset in segment
60  };
61
62  /* *
63   * Set up a normal interrupt/trap gate descriptor
64   *   - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate
65   *   - sel: Code segment selector for interrupt/trap handler
66   *   - off: Offset in code segment for interrupt/trap handler
67   *   - dpl: Descriptor Privilege Level - the privilege level required
68   *          for software to invoke this interrupt/trap gate explicitly
69   *          using an int instruction.
70   * */
71  #define SETGATE(gate, istrap, sel, off, dpl) {            \
72      (gate).gd_off_15_0 = (uint32_t)(off) & 0xffff;        \
73      (gate).gd_ss = (sel);                                 \
74      (gate).gd_args = 0;                                    \
75      (gate).gd_rsv1 = 0;                                    \
76      (gate).gd_type = (istrap) ? STS_TG32 : STS_IG32;     \
77      (gate).gd_s = 0;                                      \
78      (gate).gd_dpl = (dpl);                                 \
79      (gate).gd_p = 1;                                      \
80      (gate).gd_off_31_16 = (uint32_t)(off) >> 16;          \
81  }
```

如图，kern/mm/mmu.h中定义了两个宏用于定义或初始化门描述符。对上图宏定义的解释如下：

中断门描述符和陷阱门描述符的结构相同，均以结构体gatedesc定义，总大小为8字节。其成员变量的含义如下：

- gd_off_15_0：低四字节里低16位上的中断处理程序偏移量
- gd_ss：段选择子，位于低四字节中的高16位
- gd_args：某些参数，在中断/陷阱门描述符里不会用到，一直设为0即可，占5位
- gd_rsvl：某些保留部分，也不会用到，保持0不变，占3位
- gd_type：表示当前是什么类型的门描述符，占4位；对应的变量STS_IG32（0xE）和STS_TG32（0xF）存储在kern/mm/mmu.h中；因为中断处理时只需考虑中断门和陷阱门，因此只涉及上述两个宏定义变量
- gd_s：某系统参数，设为0即可，仅1位
- gd_dpl：中断处理过程涉及的特权级（0或3，其中中断处理程序的DPL只能是ring 0），占两位，分别用kern/mm/mmu.h中定义的DPL_KERNEL和DPL_USER来表示
- gd_p：一位标志位，如果段在内存里出现则为1，不在内存里则为0
- gd_off_31_16：高四字节里高16位上的中断处理程序偏移量

另：C语法中，变量定义里冒号后面接数字这个格式用于指定该变量的位数。

SETGATE宏将被替换为一个语句块，功能是给IDT这个结构体数组（结构体名：gatedesc）的各项成员赋值，即用于初始化门描述符，参数的意义分别是：

- gate：结合IDTR获得的地址
- istrap：0或1，0选择中断门，1选择陷阱门
- sel：中断处理程序所在段的段选择子
- off：32位偏移量，由16位的两部分拼接而成
- dpl：该门描述符对应中断处理程序的特权级

IDT初始化整体流程是先初始化内核态中断，再初始化系统调用中断，最后在IDTR寄存器中存放IDT的地址，代码如下：

```
void idt_init(void) {
     /* LAB1 YOUR CODE : STEP 2 */
     /* (1) Where are the entry addrs of each Interrupt Service Routine (ISR)?
      *     All ISR's entry addrs are stored in __vectors. where is uintptr_t __vect
      *     __vectors[] is in kern/trap/vector.S which is produced by tools/vector.
      *     (try "make" command in lab1, then you will find vector.S in kern/trap D
      *     You can use  "extern uintptr_t __vectors[];" to define this extern varia
      * (2) Now you should setup the entries of ISR in Interrupt Description Table
      *     Can you see idt[256] in this file? Yes, it's IDT! you can use SETGATE ma
      * (3) After setup the contents of IDT, you will let CPU know where is the IDT
      *     You don't know the meaning of this instruction? just google it! and che
      *     Notice: the argument of lidt is idt_pd. try to find it!
      */
        extern uintptr_t __vectors[];
        int i;
        for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i ++) {
            SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
        }
        // set for switch from user to kernel
        SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, __vectors[T_SWITCH_TOK], DPL_USER);
        // load the IDT
        lidt(&idt_pd);
}
```

- 根据提示，中断服务例程（ISR）的入口地址都存放在uintptr_t类型的__vectors数组中，该数组由tools/vectors.c生成，存放在kern/trap/vector.S中；

```
# vector table
.data
.globl __vectors
__vectors:
  .long vector0
  .long vector1
  .long vector2
  .long vector3
  .long vector4
  .long vector5
  .long vector6
  .long vector7
  .long vector8
```

- vectori即为中断向量，跳转到对应的中断向量之后，将调用kern/trap/trapentry.S中__alltraps函数保存被打断的程序的现场；
- trap.c中为IDT准备了结构体数组定义：类型为gatedesc的idt[256]，每个IDT表项就是该数组中的一项，使用SETGATE宏进行初始化；
- 中断的特权级应设为ring 0，使用/kern/mm/memlayout.h中定义的系统、用户特权级变量（DPL_KERNEL，值是0；DPL_USER，值是3）；
- IDT的内容初始化完成后，还需要将IDT的起始地址加载到IDTR寄存器里，即需调用lidt指令——在C程序里写调用的方法是使用x86.h中定义的lidt函数（该函数参数类型pseudodesc也定义在x86.h中，可以看到存放了段的大小和基址），其功能是生成内联汇编来调用lidt指令（volatile关键字的意思是编译时拒绝优化）。

```
/* Pseudo-descriptors used for LGDT, LLDT(not used) and LIDT instructions. */
struct pseudodesc {
    uint16_t pd_lim;        // Limit
    uint32_t pd_base;        // Base address
} __attribute__ ((packed));
```

```
72 static inline void
73 lidt(struct pseudodesc *pd) {
74     asm volatile ("lidt (%0)" :: "r" (pd));
75 }
```

- 在trap.c中，可以方便地将该文件里定义的pseudodesc类型结构体idt_pd实例化，作为lidt函数的参数：

```
30 static struct pseudodesc idt_pd = {
31     sizeof(idt) - 1, (uintptr_t)idt
32 };
```

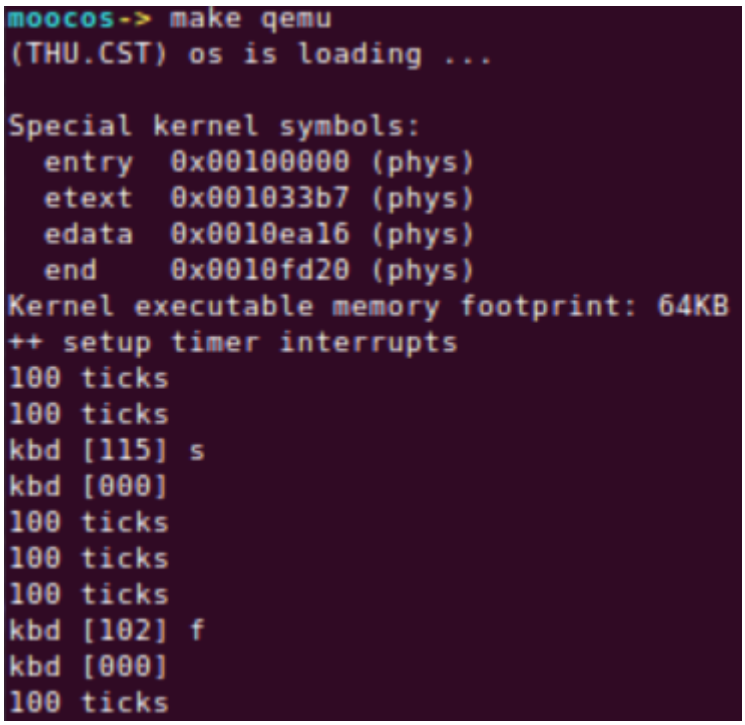
3. **请编程完善trap.c中的中断处理函数trap，在对时钟中断进行处理的部分填写trap函数中处理时钟中断的部分，使操作系统每遇到100次时钟中断后，调用print_ticks子程序，向屏幕上打印一行文字"100 ticks"。**

   trap.h当中定义了中断号；IRQ_OFFSET之后的若干编号代表硬件中断，比如IRQ_OFFSET+IRQ_TIMER表示时钟中断的中断号。

```
145 /* trap_dispatch - dispatch based on what type of trap occurred */
146 static void
147 trap_dispatch(struct trapframe *tf) {
148     char c;
149
150     switch (tf->tf_trapno) {
151     case IRQ_OFFSET + IRQ_TIMER:
152         /* LAB1 YOUR CODE : STEP 3 */
153         /* handle the timer interrupt */
154         /* (1) After a timer interrupt, you should record this even
155          * (2) Every TICK_NUM cycle, you can print some info using
156          * (3) Too Simple? Yes, I think so!
157          */
158     ticks++;
159     if(ticks % TICK_NUM == 0)
160         print_ticks();
161         break;
```

   按照给出的提示填充trap_dispatch函数，使用全局变量ticks记录已发生的时钟中断数目；该中断每发生每100（TICK_NUM）次调用一遍print_ticks函数打印字符串即可。

重新编译并运行整个系统（lab1目录下执行make qemu），可以看到打印出了时钟中断信息，按下的键也会显示在屏幕上：



函数调用关系：

trap->trap_dispatch->

遇到的问题：

1. 起初对于段机制和中断处理流程了解不清晰，查阅资料后基本明确了相关内容。

参考资料

ifdef-endif 条件编译：https://blog.csdn.net/andylanzhiyong/article/details/78575354

两张图看懂GDT、GDTR、LDT、LDTR的关系：https://blog.csdn.net/Six_666A/article/details/80634972

门描述符：https://blog.csdn.net/chen1540524015/article/details/73817554

中断发生时，CPU通过8259A选择片收到中断，然后通过IDTR寄存器找到IDT表，读取中断或异常向量i对应的表项（第i个门），跳转到对应的处理例程中。

在程序中写入 int n 指令可以产生n号中断和异常；这是用户代码带来的中断，因此实际中断发生时将在原始基础上附加一个特权级检查，门（中断服务例程入口）的DPL必须<=CPL。

当中断和异常发生时, NMI和异常的向量是由处理器自动给出的; 硬件的向量是由I/O中断控制器芯片送给处理器的； 软中断的向量是由指令中的操作数给出的。

计算机需要对内存分段，以分配给不同的程序使用（类似于硬盘分页）。在描述内存分段时，需要有如下段的信息：1.段的大小；2.段的起始地址；3.段的管理属性（禁止写入/禁止执行/系统专用等）。

vectors.S中定义了256个中断向量。

mmu.h中定义了改变标志位的宏

IDT表可以驻留在线性地址空间的任何地方，处理器使用IDTR寄存器来定位IDT表的位置。

vectors.c中代码功能是打印汇编程序，编译后生成vectors.S文件

首先找到中断向量入口，即中断处理程序的起始地址，然后使用mmu.h中的SETGATE宏填充idt数组内容，最后使用lidt指令告知CPU中断向量表的位置。源码见附件。

32位汇编中16位段寄存器(CS、DS、ES、SS、FS、GS)中不再存放段基址,而 是段描述符在段描述符表中的索引值,D3-D15位是索引值,D0-D1位是优先级(RPL)用于特权检查,D2位是描述符表引用指示位TI,TI=0指 示从全局描述表GDT中读取描述符，TI=1指示从局部描述符中LDT中读取描述符。这些信息总称段选择符(段选择子).

段寄存器16位，此外还有一些隐藏的缓存部分。32位汇编中段寄存器存放段选择子（而非段基址），段选择子的0-1位表示特权级；第2位是指示位，0表示从GDT中读取描述符，1表示从LDT中读取描述符；3-15位是索引值，在GDT/LDT中查表时会用到。

IDTR寄存器有48位，后32位表示段基址，前16位表示段长度（limit）。

NMI(非maskable中断：Non-maskable interrupt)，顾名思义是不能屏蔽的中断。在正常中断中，即使发生中断因素，也可以设置为屏蔽中断而不接受中断(称为Maskable interrupt中断)。但是，在NMI的情况下，不能屏蔽，在NMI的中断因素发生的情况下，一定会接受中断。当系统发生致命故障时，使用NMI。当发生NMI的因素时，不接受任何其他中断，启动紧急情况中断处理程序，并运行紧急情况处理程序软件。

在保护模式下，中断描述符表（IDT）中的每个表项由8个字节组成，其中的每个表项叫做一个门描述符（Gate Descriptor）， “门”的含义是指当中断发生时必须先访问这些“门”，能够“开门”（即将要进行的处理需通过特权检查，符合设定的权限等约束）后，然后才能进入相应的处理程序。而门描述符则描述了“门”的属性（如特权级、段内偏移量等）。在IDT中，可以3种类型的系统段描述符：中断门描述符、陷阱门描述符、任务门描述符

扩展练习 Challenge1(需要编程)

> 扩展proj4,增加syscall功能，即增加一用户态函数（可执行一特定系统调用：获得时钟计数值），当内核初始完毕后，可从内核态返回到用户态的函数，而用户态的函数又通过系统调用得到内核态的服务（通过网络查询所需信息，可找老师咨询。如果完成，且有兴趣做代替考试的实验，可找老师商量）。需写出详细的设计和分析报告。完成出色的可获得适当加分。

首先为了完成特权级转换，需要了解这些知识：

- int iret在不同情况下的执行步骤
- 特权级检查

阅读实验指导书，kern/init/init.c和kern/trap/trap.c文件，不难发现这个challenge需要我们完成以下四个内容：

- kern/init/init.c　中的 switch_to_user
- kern/init/init.c　中的 switch_to_kernel
- kern/trap/trap.c 中的 case T_SWITCH_TOU #to user
- kern/trap/trap.c 中的 case T_SWITCH_TOK #to kernel

因为在调用关系中是 init.c调用trap.c，所以先从init.c中入手。

先来看switch_to_user。很好，什么也没有，满足我对于难度的要求。

```
static void
lab1_switch_to_user(void) {
    //LAB1 CHALLENGE 1 : TODO
}
```

没有东西只能白手起家，还能咋地。先把写好的代码贴出来，之后再进行详细的解释

```
//  init.c
static void
```

```
lab1_switch_to_user(void) {
    //LAB1 CHALLENGE 1 : TODO
    asm volatile(
            "sub $0x8,%%esp \n"
            "int %0 \n"
            "movl %%ebp, %%esp \n"
            :
            :"i"(T_SWITCH_TOU)
        );
}
static void
lab1_switch_to_kernel(void) {
    //LAB1 CHALLENGE 1 :  TODO
    asm volatile(
            "int %0 \n"
            "movl %%ebp, %%esp \n"
            :
            :"i"(T_SWITCH_TOK)
        );
}
```

```
// trap.c
    case T_SWITCH_TOU:
        if(tf->tf_cs != USER_CS)    //检查是不是用户态，不是就操作
        {
                cprintf("...to user\n");
                // 设置用户态对应的cs,ds,es,ss四个寄存器
                tf->tf_cs = USER_CS;
                tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
                // 为用户态带来可以I/O的快乐
                tf->tf_eflags |= FL_IOPL_MASK;
        }
        break;

    case T_SWITCH_TOK:
        if(tf->tf_cs != KERNEL_CS)  //检查是不是内核态，不是就操作
        {
                cprintf("...to kernel\n");
                // 设置内核态对应的cs,ds,es三个寄存器
                tf->tf_cs = KERNEL_CS;
                tf->tf_ds = tf->tf_es = KERNEL_DS;
                // 剥夺用户态可以使用I/O的快乐
                tf->tf_eflags &= ~FL_IOPL_MASK;
        }
        break;
```

一开始我以为user_to_kernel和kernel_to_user应该没有什么区别，但这个challenge1不愧是个challenge。其中的区别在中断发生的压栈状况有关系。

中断可以发生在任何一个特权级别下，但是不同的特权级处理器使用的栈不同，如果涉及到特权级的变化，需要对SS和ESP寄存器进行压栈。性质如下：

- 当低特权级向高特权级切换的压栈(**用户态到内核态**)

需要判断是否能访问这个目标段描述符，要做的就是将找到中断描述符时的CPL与目标段描述符的DPL进行比较。当CPL特权级比DPL低(CPL>DPL)时，要往高特权级栈转移，也就是说要恢复旧栈，因此处理器临时保存旧栈的SS和ESP，然后加载新的特权级和DPL相同的段到SS和ESP中，把旧栈的SS和ESP压入新栈

- 当无特权级转化时的压栈(**内核态到用户态**)

  理论上来说从内核态到用户态也需要对栈进行切换，不过在lab1中并没有完整实现对物理内存的管理，而GDT中的每一个段除了对特权级的要求以外都一样，所以只需要修改一下权限就可以实现了。这也导致这个时候不会压栈，我们需要手动压栈(体现在lab1_switch_to_user中的sub $0x8,%%esp)。

不过在trap.c中的实现比较雷同，把对应的tf指针修改为对应态的内容就行。

遇到的一些麻烦和问题

1. **关于int和iretz**

   这俩东西可以说是中断的灵魂，如果搞不懂这个真的没法做实验。用这两个中断过程来举例

   1. 中断触发后，处理器根据中断向量号找到对应的中断描述符，然后拿**中断发生时的CPL**和**中断描述符**中的**段选择子**对应的**DPL**做对比，如果发现CPL权限比DPL低(CPL数值更大)时，将旧栈压入新栈，具体表现就是将ss_old和esp_old压入ss_new和esp_new中。用户态到内核态的栈切换由TSS和硬件实现
   2. **在用户态到内核态的切换过程中**，依次压入(高地址)ss,esp,eflags,cs,eip,errorno(低地址)等参数；**在内核态到用户态的切换过程中**，依次压入(高地址)eflags,cs,eip,errorno(低地址)等参数
   3. **在用户态到内核态的切换过程中**，不用给空间，直接用int %0触发中断。这个int %0对应的是我们的输入(T_SWITCH_TOK)，调用前后函数后我们的栈帧如下

      > //调用前
      >
      > (高地址)user_ss,野指针,eflags,user_cs,eip,errorno,trapno,user_ds,user_es(低地址)
      >
      > // 调用后
      >
      > (高地址)user_ss,野指针,eflags,kernel_cs,eip,errorno,trapno,kernel_ds,kernel_es(低地址)

      在内核态到用户态的切换过程中，先通过sub $0x8,%%esp给8B的空间，之后同样用int %0触发中断，此时对应的就是(T_SWITCH_TOU)，调用前后的栈帧如下

      > //调用前
      >
      > (高地址)野指针,野指针,eflags,kernel_cs,eip,errorno,trapno,kernel_ds,kernel_es(低地址)
      >
      > // 调用后
      >
      > (高地址)user_ss,野指针,eflags,user_cs,eip,errorno,trapno,user_ds,user_es(低地址)

   4. 执行完中断程序之后，通过**iret**返回，依次弹出对应段选择子从而实现对栈的切换。在弹出eip和cs之后，根据cs中的RPL判断是否需要继续弹出。**也就是说，如果要返回到特权级更低的代码，就要弹出ss和esp。**

      在用户态到内核态的切换过程中，栈中的CS是kernel_cs，DPL=0，当前的CPL=3，代码不会返回到更低的特权级，所以不弹出esp和ss。**但是栈所在的段已经发生了变化**，也就是SS已经发生了变化：在用户态下中断会导致user_ss和user_esp被压入新的内核栈，所以最开始的ss就是kernel_ss。这也是为什么在TOK中不用设置tf_ss。

      在内核态到用户态的切换过程中，栈中的CS是user_cs，DPL=3，当前的CPL=0，代码会返回到更低的特权级，所以会弹出esp和ss，这个时候野指针被pop到esp，user_ss被弹出到ss，实现了栈段的切换，内核切换到了用户栈。

5. 还有最后一句话movl %%ebp, %%esp。同样在两种情况下看

   在用户态到内核态的切换过程中，要回收user_ss,user_esp，所以通过这句话让esp指向ebp，并且顶掉原来储存在这个位置的user_esp，而4中我们知道了这个时候的user_ss实际上是kernel_ss，所以不用管他。

   在内核态到用户态的切换过程中，此时的esp是野指针，是一个内存的初始值，我们需要让他指向ebp。那怎么给ebp呢？很巧妙，在sub $0x8,%%esp这句话执行之后，栈帧状态是这样的

   > (高地址)野指针，野指针(低地址)

   但其实原来的ebp刚好就在上面，实际上是这样

   > (高地址)原ebp，(ebp指向)，野指针，野指针(低地址)

   所以这句话就能帮助我们让esp的野指针指向ebp

2. **在用户态和内核态的切换时，虽然eip没变，但是段在变，为什么还能正常运行?**

   一开始没想明白，后面脑子突然上线：哦，**我们现在是保护模式**

   所以我们的CS不再是直接的代码段，而是段选择子，并不是一个实际的物理段地址而是一个索引，通过这个索引去查这个段具体的物理地址。

   具体的格式如下

   > 格式为：【索引(13)|TI(1)|RPL(2)】
   >
   > 索引：GDT表中有8K个表项(2^13=8k)
   >
   > TI： 0-GDT 1-LDT
   >
   > RPL：00-kernel，11-user
   >
   > 内核态下的8 = 【00...01|0|00】
   >
   > 用户态下的1b = 【00...11|0|11】

   这里面影响地址的只有索引，他俩的索引一个是1，3。索引和GDT表有关，那么我们看看GDT表的相关内容

```
// kern/mm/pmm.c
static struct segdesc gdt[] = {
    SEG_NULL,
    [SEG_KTEXT] = SEG(STA_X | STA_R, 0x0, 0xFFFFFFFF, DPL_KERNEL),
    [SEG_KDATA] = SEG(STA_W, 0x0, 0xFFFFFFFF, DPL_KERNEL),
    [SEG_UTEXT] = SEG(STA_X | STA_R, 0x0, 0xFFFFFFFF, DPL_USER),
    [SEG_UDATA] = SEG(STA_W, 0x0, 0xFFFFFFFF, DPL_USER),
    [SEG_TSS]   = SEG_NULL,
};
```

   1对应的是[SEG_KTEXT]，内核段，ok，3对应的是[SEG_UTEXT]，用户段，也ok。这里出现的新东西是 segdesc 和 SEG，再去看看这俩到底是个啥

```
// kern/mm/mmu.h
struct segdesc {
    unsigned sd_lim_15_0 : 16;        // low bits of segment limit
    unsigned sd_base_15_0 : 16;        // low bits of segment base address
    unsigned sd_base_23_16 : 8;        // middle bits of segment base
address
    unsigned sd_type : 4;            // segment type (see STS_ constants)
    unsigned sd_s : 1;                // 0 = system, 1 = application
```

```
    unsigned sd_dpl : 2;            // descriptor Privilege Level
    unsigned sd_p : 1;                // present
    unsigned sd_lim_19_16 : 4;        // high bits of segment limit
    unsigned sd_avl : 1;            // unused (available for software use)
    unsigned sd_rsv1 : 1;             // reserved
    unsigned sd_db : 1;               // 0 = 16-bit segment, 1 = 32-bit
segment
    unsigned sd_g : 1;                // granularity: limit scaled by 4K
when set
    unsigned sd_base_31_24 : 8;       // high bits of segment base address
};

#define SEG(type, base, lim, dpl)                       \
    (struct segdesc){                                   \
        ((lim) >> 12) & 0xffff, (base) & 0xffff,        \
        ((base) >> 16) & 0xff, type, 1, dpl, 1,           \
        (unsigned)(lim) >> 28, 0, 0, 1, 1,                \
        (unsigned) (base) >> 24                           \
    }
```

我们看到SEG括号内的第二个参数base，都是0x0，破案了，**他们虽然表面上选择子一直在换，但是他们所指向的实际物理段基址并没有变，是一样的。**

扩展练习 Challenge2(需要编程)

> 用键盘实现用户模式内核模式切换。具体目标是：“键盘输入3时切换到用户模式，键盘输入0时切换到内核模式”。 基本思路是借鉴软中断(syscall功能)的代码，并且把trap.c中软中断处理的设置语句拿过来。

在Challenge1中我们其实已经实现了用户模式和内核模式的相互切换，所需要的只是增加一个用键盘输入来控制切换的功能。我们找到控制状态切换的trap.c文件中的trap_dispatch函数。他已经很贴心的给我们写了一个case IRQ_OFFSET + IRQ_KBD:(键盘输入情况)

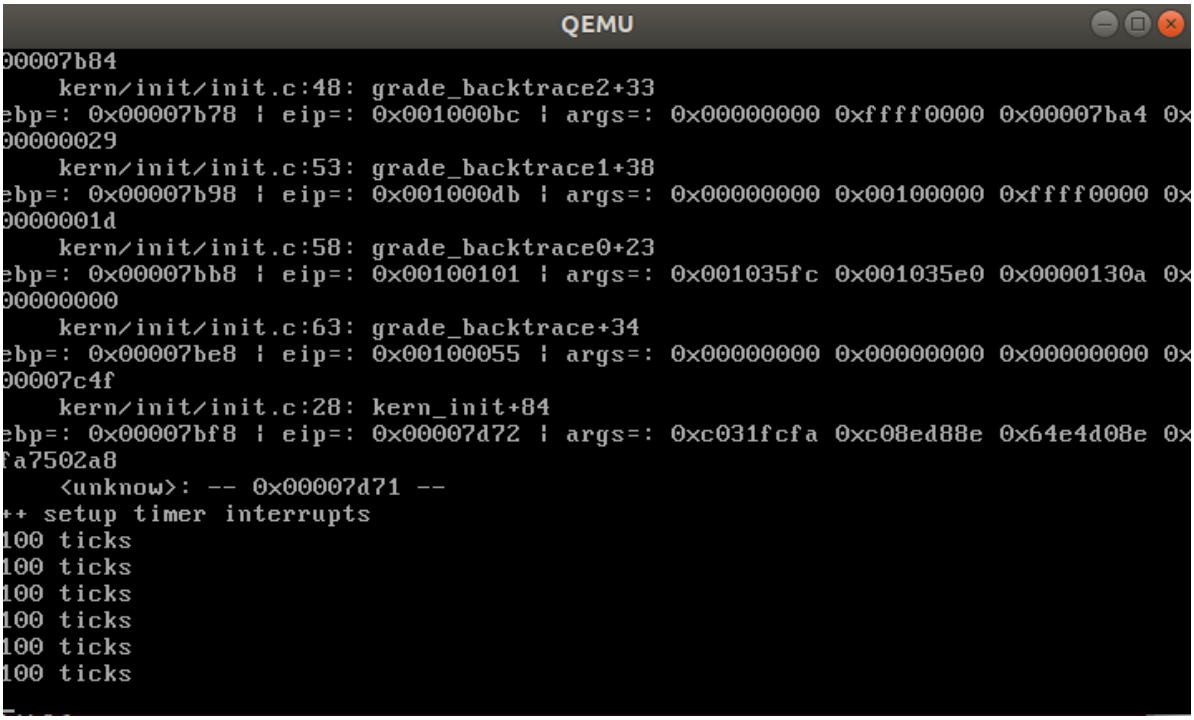
很自然的能够想到，用**if-else**结构就可以实现一个控制。那么到此编程的思路已经十分明确了。直接贴出我们的代码

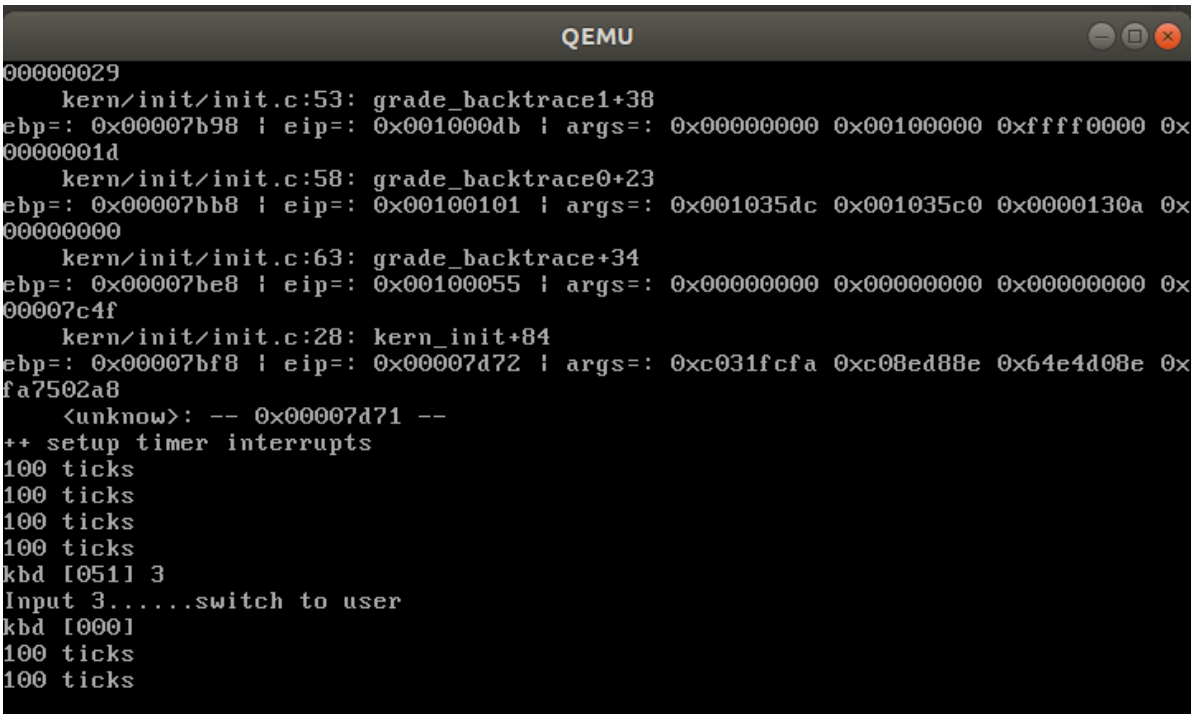
```
case IRQ_OFFSET + IRQ_KBD:
        c = cons_getc();
        cprintf("kbd [%03d] %c\n", c, c);
        if(c == '0' && (tf->tf_cs & 3) != 0)
        {
                cprintf("Input 0......switch to kernel\n");
                tf->tf_cs = KERNEL_CS;
                tf->tf_ds = tf->tf_es = KERNEL_DS;
                tf->tf_eflags &= ~FL_IOPL_MASK;
        }
        else if (c == '3' && (tf->tf_cs & 3) != 3)
        {
                cprintf("Input 3......switch to user\n");
                tf->tf_cs = USER_CS;
                tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
                tf->tf_eflags |= FL_IOPL_MASK;
        }
        break;
```

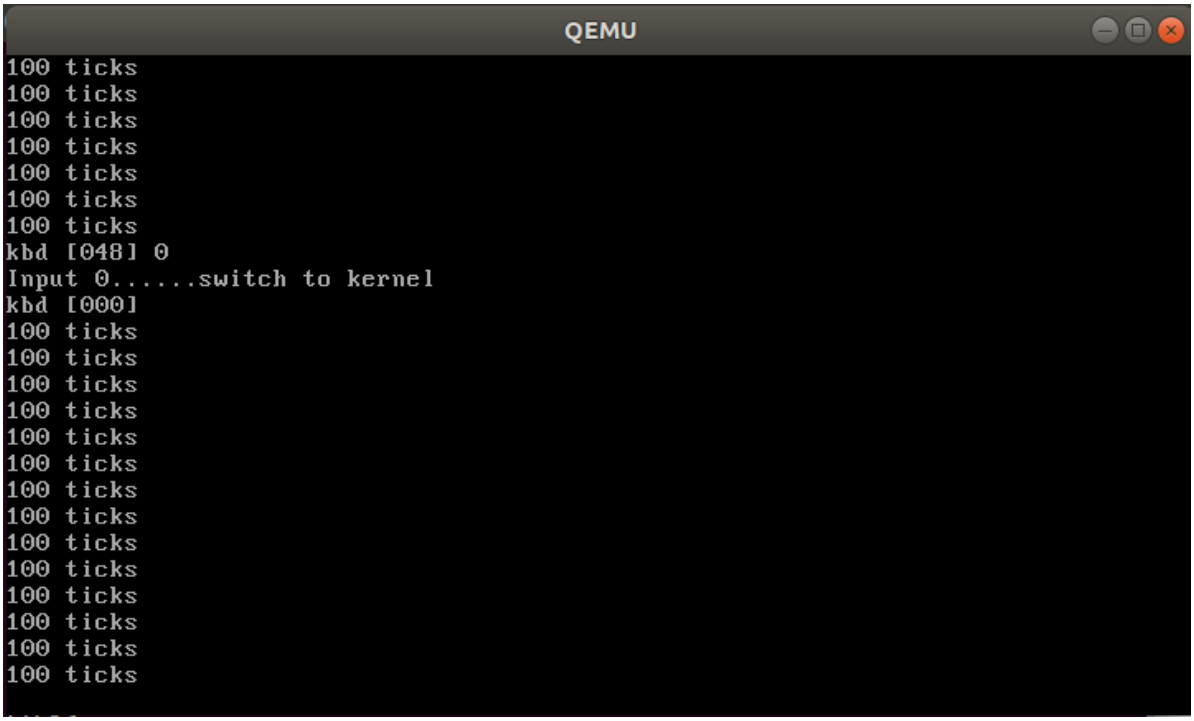
完成了所有的实验和challeng之后，使用

```
$ make qemu
```

执行出来的程序应该出现如下情况







最后make grade查看输出

```
yyf@ubuntu:~/ucore_os-master/labcodes/lab1$ make grade
Check Output:             (2.5s)
  -check ring 0:                                   OK
  -check switch to ring 3:                         OK
  -check switch to ring 0:                         OK
  -check ticks:                                    OK
Total Score: 40/40
```

同时，也会生成一些不可见文件