练习一：通过make生成执行文件的过程

1.操作系统镜像文件ucore.img的生成

在Makefile文件中可以看到ucore.img是如何生成的，且ucore.img依赖于kernel和bootblock，这两个文件的生成同样需要进行分析。

ucore.img的生成

Makefile中生成ucore.img的部分如下：

```
# create ucore.img
UCOREIMG    := $(call totarget,ucore.img)           #定义
$(UCOREIMG): $(kernel) $(bootblock)                 #依赖文件
    $(V)dd if=/dev/zero of=$@ count=10000
    $(V)dd if=$(bootblock) of=$@ conv=notrunc
    $(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc
$(call create_target,ucore.img)
```

首先为目标文件ucore.img定义变量名UCOREIMG，然后指定其依赖文件为kernel和bootblock为变量名的文件。接下来使用dd命令完成ucore.img的生成。

> dd命令及其他使用的参数如下：

- if=文件名：输入文件名，默认为标准输入。即指定源文件。
- of=文件名：输出文件名，默认为标准输出。即指定目的文件。
- seek=blocks：从输出文件开头跳过blocks个块后再开始复制。
- count=blocks：仅拷贝blocks个块。
- /dev/zero：提供无限的空字符(NULL, ASCII NUL, 0x00)。常见用法是产生一个特定大小的空白文件。
- conv=<关键字>，notrunc：不截短输出文件

根据这些参数，dd命令生成ucore.img的过程具体为：

```
#含10000个512字节（默认）的块的空白UCOREIMG
$(V)dd if=/dev/zero of=$@ count=10000
#将bootblock放入UCOREIMG的第一个块
$(V)dd if=$(bootblock) of=$@ conv=notrunc
#跳过第一个块，将kernel放入UCOREIMG
$(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc
```

kernel的生成

Makefile中kernel生成的部分如下：

```
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)
```

kernel的生成需要kernel.ld以及KOBJS变量指向的文件（kernel libs）。其中kernel.ld是链接脚本。

$(V)(LD) (LDFLAGS) -T tools/kernel.ld -o @ $(KOBJS) 将KOBJS指向的文件链接生成kernel文件，-T表示使用kernel.ld替代默认链接脚本。

@(OBJDUMP) -S @ > $(call asmfile,kernel) 表示使用objdump得到kernel反汇编代码，-S表示源代码和汇编代码共同显示，并重定向保存到kernel.asm文件中。

@(OBJDUMP) -t @ | $(SED) '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > (call symfile,kernel) 表示将文件的符号表保存到kernel.sym文件中，-t表示打印出文件的符号表表项，|表示使用管道将符号表作为SED的输入，最后保存到kernel.sym文件中。（SED命令用于处理文本文件）

使用make V= 命令可以查看生成kernel的指令，可以看到所有进行链接的文件

```
+ ld bin/kernel
ld -m    elf_i386 -nostdlib -T tools/kernel.ld -o bin/kernel  obj/kern/init/init.o
obj/kern/libs/stdio.o obj/kern/libs/readline.o obj/kern/debug/panic.o
obj/kern/debug/kdebug.o obj/kern/debug/kmonitor.o obj/kern/driver/clock.o
obj/kern/driver/console.o obj/kern/driver/picirq.o obj/kern/driver/intr.o
obj/kern/trap/trap.o obj/kern/trap/vectors.o obj/kern/trap/trapentry.o
obj/kern/mm/pmm.o  obj/libs/string.o obj/libs/printfmt.o
```

可以看到kernel的生成需要链接init.o，stdio.o，readline.o等文件。使用make V=命令也可以看到这些文件由c程序经编译产生.o文件的过程。以init.o的生成为例：

```
+ cc kern/init/init.c
gcc -Ikern/init/ -march=i686 -fno-builtin -fno-PIC -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
```

其他.o文件生成类似。

> 参数说明：

- -m elf_i386：表示模拟i386的链接器来完成.o文件链接为可执行文件的过程
- -nostdlib ： 表示不链接任何标准库，kernel不需要使用标准库
- -I：添加搜索头文件的路径
- -march=i686：根据目标架构进行优化
- -fno-builtin：不使用c语言的内建函数，避免自定义函数与内建函数冲突的问题
- -fno-PIC：不使用位置无关代码
- -Wall：显示所有警告信息
- -ggdb：专门为gdb产生调试信息
- -m32：生成32位机的机器代码
- -gstabs：以stabs格式输出调试信息，不包括gdb
- -nostdinc: 不搜索默认路径头文件
- -fno-stack-protector: 禁用堆栈保护

bootblock的生成

Makefile中有关bootblock生成的部分如下：

```
# 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 $@
    #链接生成bootblock.o文件
    $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ -o $(call toobj,bootblock)
    #将文件反汇编保存到bootblock.asm文件中
    @$(OBJDUMP) -S $(call objfile,bootblock) > $(call asmfile,bootblock)
    #将bootblock.o使用OBJCOPY复制到bootblock.out中
    @$(OBJCOPY) -S -O binary $(call objfile,bootblock) $(call outfile,bootblock)
    #使用sign生成bootblock
    @$(call totarget,sign) $(call outfile,bootblock) $(bootblock)
$(call create_target,bootblock)
```

> 参数说明：

- -N：将代码段和数据段设置为可读可写
- -e：设置入口
- $^：表示所有依赖对象
- -Ttext：设置起始地址（0X7C00）
- -S：移除符号和重定位信息

bootblock的生成依赖于sign和bootfiles文件，执行make V=可以找到bootblock生成的部分，发现bootfiles包括bootmain.o和bootasm.o，可以找到bootmain.o和bootasm.os及sign的生成部分。

```
+ cc boot/bootasm.S
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
+ cc boot/bootmain.c
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
+ cc tools/sign.c
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
+ ld bin/bootblock
ld -m    elf_i386 -nostdlib -N -e start -Ttext 0x7C00 obj/boot/bootasm.o
obj/boot/bootmain.o -o obj/bootblock.o
'obj/bootblock.out' size: 496 bytes
//-Os参数：为减小文件大小进行优化
```

首先生成bootmain.o，bootasm.o，sign，然后使用sign生成bootblock。

综上所述，ucore.img的生成过程如下：

- 生成kernel所需要的.o文件，将这些文件链接得到kernel
- 生成bootmain.o，bootasm.o链接得到bootblock.out，使用sign将bootblock.out转换为bootblock
- 创建一个有10000个512字节的块的空文件，将bootblock放入第一个块，kernel放入剩下的块中。

2.符合规范的硬盘主引导扇区的特征

符合规范的硬盘主引导扇区的特征为：大小为512字节；最后两个字节是0x55和0xAA，这是合法主引导扇区的标志。

bootblock由sign生成，在sign.c中也可以看到这两个特征：

```
if (st.st_size > 510) {
        fprintf(stderr, "%lld >> 510!!\n", (long long)st.st_size);
//bootblock小于510字节
        return -1;
}
buf[510] = 0x55;                                                    //最后
两个字节为0x55,0xAA
buf[511] = 0xAA;
```

练习二：使用qemu执行并调试lab1中的软件

1.从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行。

启动后的第一条指令和BIOS完成的工作

当计算机启动时，寄存器CS，EIP中的值将被初始化，EIP=0xfff0，而CS的selector=0xf000，base=ffff0000。CS.base+EIP=0xffffffff0，这就是第一条指令的位置。（段寄存器分为可见部分和隐藏部分（描述符高速缓存器），此处CS.base是加电后直接设置的。）而这个位置的指令为一条跳转指令，跳转到0xf000e05b这个位置，这个位置就是BIOS开始的地方。BIOS将会完成硬件自检及初始化，创建中断向量表等工作，并读取第一扇区（主引导扇区或启动扇区）到内存地址0x7c00，接下来的工作交给bootloader完成。

将lab1/tools/gdbinit文件修改为如下内容：

```
set architecture i8086
target remote :1234
```

接下来在该目录下make debug，弹出qemu及gdb调试窗口，可以查看第一条指令，并使用si单步跟踪BIOS的执行，使用x/i命令可以查看执行的具体汇编指令。

```
0x0000fff0 in ?? ()
(gdb) si
0x0000e05b in ?? ()
(gdb) x/i 0xffffffff0
   0xffffffff0:  ljmp   $0x3630,$0xf000e05b
```

2.在初始化位置0x7c00设置实地址断点,测试断点正常

查看lab1init文件，内容为调试命令

```
file bin/kernel                     //加载
target remote :1234                 //链接qemu
set architecture i8086              //设定架构为i8086
b *0x7c00                           //断点设置在0x7c00
continue
x /2i $pc                           //查看两条指令
```

使用make lab1-mon命令进入调试，终端显示如下：

```
breakpoint 1, 0x00007c00 in ?? ()
=> 0x7c00:    cli
   0x7c01:    cld
(gdb)
```

在0x7c00处设置断点正常，可以从这里开始正常调试bootloader。

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

从0x7c00开始x/15i显示接下来执行的15条指令，指令如下：

```
(gdb) x/15i 0x7c00
=> 0x7c00:    cli
   0x7c01:    cld
   0x7c02:    xor    %eax,%eax
   0x7c04:    mov    %eax,%ds
   0x7c06:    mov    %eax,%es
   0x7c08:    mov    %eax,%ss
   0x7c0a:    in     $0x64,%al
   0x7c0c:    test   $0x2,%al
   0x7c0e:    jne    0x7c0a
   0x7c10:    mov    $0xd1,%al
   0x7c12:    out    %al,$0x64
   0x7c14:    in     $0x64,%al
   0x7c16:    test   $0x2,%al
   0x7c18:    jne    0x7c14
   0x7c1a:    mov    $0xdf,%al
```

bootasm.S中可以找到和以上相同的汇编代码：

```
.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
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
    ......
```

在obj文件夹下找到bootblock.asm，其中的代码同样和以上代码相同：

```
code16                                             # Assemble for 16-bit mode
    cli                                             # Disable interrupts
    7c00:    fa                        cli
    cld                                             # String operations increment
    7c01:    fc                        cld

    # Set up the important data segment registers (DS, ES, SS).
    xorw %ax, %ax                                   # Segment number zero
    7c02:    31 c0                     xor    %eax,%eax
    movw %ax, %ds                                   # -> Data Segment
    7c04:    8e d8                     mov    %eax,%ds
    movw %ax, %es                                   # -> Extra Segment
    7c06:    8e c0                     mov    %eax,%es
    movw %ax, %ss                                   # -> Stack Segment
    7c08:    8e d0                     mov    %eax,%ss
    ......
```

即0x7c00处开始执行的汇编指令与bootasm.S及bootblock.asm中的代码是相同的。

4.找一个bootloader或内核中的代码位置，设置断点并进行测试。

在0x7c14处设置断点并进行调试：

```
(gdb) b *0x7c14
Breakpoint 2 at 0x7c14
(gdb) c
Continuing.

Breakpoint 2, 0x00007c14 in ?? ()
(gdb) x/5i $pc
=> 0x7c14:    in     $0x64,%al
   0x7c16:    test   $0x2,%al
   0x7c18:    jne    0x7c14
   0x7c1a:    mov    $0xdf,%al
   0x7c1c:    out    %al,$0x60
(gdb)
```

练习三：分析bootloader进入保护模式的过程

bootloader完成的最重要的两个工作为：

- **实模式到保护模式的切换**
- **读取硬盘并加载OS**

分析bootloader进入保护模式的过程，需要了解以下内容：

实模式与保护模式

BIOS完成需要完成的初始化等工作后，将控制权交给bootloader，此时操作系统处于实模式(16位)，可访问的物理内存空间不能超过1MB，转换为保护模式后，全部32根地址线有效，可以采用分段机制或分页机制，寻址4G的物理地址空间。

分段机制和GDT表

分段机制是管理虚拟内存空间的一种机制，只有在操作系统进入保护模式后才可以使用。分段机制将内存划分成以起始地址和长度限制这两个参数表示的内存块，这些内存块就称之为段。每个段的信息存储在**段描述符**中，包括段的基址，界限，类型和特权值等信息。而段描述符存放在**全局描述符表**GDT中。全局描述符表的起始地址存放在GDTR寄存器中，通过起始地址+索引找到一个段描述符（即段的信息），其中的索引被称为**段选择子**。逻辑地址由段选择子selector（对应一个段寄存器）和段偏移offset组成，以对应的段寄存器中的段选择子为索引就可以找到表中的

段描述符，段的基址+偏移量就可以得到线性地址，在不开启分页机制的情况下，这个线性地址就是物理地址，如果开启分段机制则还需要将线性地址转换为物理地址。

除了全局描述符表GDT外，还有局部描述符表LDT同样用于存放段描述符，本实验只使用了全局描述符表。

A20 GATE

A20地址线的存在与8086的回绕有关。在8086中，采取段地址+偏移量的机制寻址，8086的数据位宽为16位，将段地址左移四位加上偏移量就可以实现1MB空间的寻址，这种寻址方式下的最大地址为0x10ffefh，但实际8086只有1MB的空间，因此地址超过1MB时会发生回绕。在此后的计算机中，如果地址空间及地址总线的寻址能力超过了1MB，就不需要回绕了，但这会造成不能向下兼容。因此，采用A20地址线控制来模仿回绕的特征，当A20控制关闭时不可访问超过1MB的空间，打开时就可以寻址访问全部地址空间了。

基于以上知识，对bootasm.S中bootloader的代码进行详细分析，从实模式转换到保护模式的过程如下：

1.关中断，清零段寄存器

```
.globl start
start:
.code16                                             # 16位实模式
    cli                                             # 关中断
    cld                                             # DF置0（串操作时使地址向高地址方向增加）
    xorw    %ax,%ax                                 # 0
    movw    %ax,%ds                                 # 数据段寄存器
    movw    %ax,%es                                 # 扩展段寄存器
     movw    %ax,%ss                                # 栈段寄存器
```

2.使能A20

A20的控制是通过8042键盘控制器实现的（节省硬件成本，A20 GATE本身与键盘控制无关）。

8042有三个内部端口，inputport（P1）,outputport（P2），testport，其中P2的bit1用于控制A20GATE，将这一位设置位1即可开启A20，如何将这一位进行设置需要简单了解8042。

> output端口操作

- 读Output Port：向64h发送0d0h命令，然后从60h读取Output Port的内容
- 写Output Port：向64h发送0d1h命令，然后向60h写入Output Port的数据

> 8042的四个8位寄存器

- 只写的inputbuffer
- 只读的outputbuffer
- 只读的status register
- 可读写的control register

> 8042两个端口的读写操作

- 读60h端口，读output buffer
- 写60h端口，写input buffer
- 读64h端口，读Status Register
- 操作Control Register，首先要向64h端口写一个命令（20h为读命令，60h为写命令），然后根据命令从60h端口读出Control Register的数据或者向60h端口写入Control Register的数据（64h端口还可以接受许多其它的命令）。

当准备向8042的输入缓冲区里写数据时，可能里面还有其它数据没有处理，需要等待数据缓冲区中没有数据以后，才能进行操作。

```
    seta20.1:
      inb     $0x64,%al               # 等待输入缓冲区空闲
      testb    $0x2,%al               # status register的bit1表示input register
有数据
      jnz     seta20.1                # 如果缓冲区非空则跳转，继续等待
      movb    $0xd1,%al               # 写output端口的命令
      outb    %al,$0x64               # 64端口接收命令
    seta20.2:
      inb     $0x64,%al               # 等待输入缓冲区空闲
      testb   $0x2,%al
      jnz     seta20.2
      movb    $0xdf,%al
      outb    %al,$0x60               # 将A20打开（bit1为1）
```

3.初始化全局描述符表

初始化GDT表，需要使用lgdt指令，将全局描述符表的开始位置和界限（gdt的大小-1）装载到GDTR寄存器当中，这样就完成了全局描述符表的初始化。

```
    lgdt gdtdesc
```

gdtdesc指出了全局描述符表起始位置在gdt处，gdt处存放了三个段描述符，每个8字节，第一个是NULL段描述符，表示全局描述符表的开始，接下来是代码段和数据段的段描述符，其中STA_X，STA_R，STA_W分别表示可写，可读，可写可读属性。同时这两个段的基址和界限是相同的，这说明实际上无论选择哪个段，寻址的方式都是相同的。

```
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
```

4.使能并进入保护模式

保护模式的启动位是CR0控制寄存器的第0位，将这一位置1，就可以使能保护模式了。接下来使用ljmp指令，跳转进入保护模式。80386在执行长跳转指令时，会重新加载PROT_MODE_CSEG的值（即0x8，内核代码段选择子）到CS段寄存器中，同时把$protcseg（保护模式代码的起始位置）的值赋给EIP，CS的值作为全局描述符表的索引来找到对应的代码段描述符，结合eip的值，就可以跳转进入保护模式的protcseg处了。

```
    movl    %cr0, %eax
    orl     $CR0_PE_ON, %eax                        # CR0_PE_ON=0x1
    movl    %eax, %cr0                              # CR0保护模式标识置1
    ljmp    $PROT_MODE_CSEG, $protcseg
```

进入保护模式后，首先设置了段寄存器的值，并开辟了栈空间，0x7c00即start位置前的空间没有用到，使用这部分空间作为栈空间，将栈顶指针设为0x7c00。最后调用bootmain，在bootmain中将会完成OS的读取和加载的后续工作。

```
.code32                                                 # Assemble for 32-bit mode
protcseg:
    # Set up the protected-mode data segment registers
    movw $PROT_MODE_DSEG, %ax                           # 数据段
    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
    # 开辟栈空间
    movl $0x0, %ebp
    movl $start, %esp
    call bootmain
```

练习四：分析bootloader加载ELF格式的OS的过程

BootLoader让CPU进入保护模式后，接下来的工作就是从硬盘上加载并运行OS。在bootmain()中会调用读取硬盘的函数和加载elf文件，完成从硬盘上加载并运行OS。

分析bootloader加载OS，需要了解硬盘扇区读取和elf格式。

1.硬盘扇区读取

bootloader访问硬盘的方式是LBA模式的PIO（Program IO）方式，所有的IO操作通过CPU访问硬盘的IO地址寄存器完成（LBA模式是硬盘的逻辑块寻址模式，硬盘的IDE会把柱面，磁头等参数形成逻辑地址转换为实际物理地址）。一般主板有2个IDE通道，每个通道可以接2个IDE硬盘。访问第一个硬盘的扇区可设置IO地址寄存器0x1f0-0x1f7实现，具体参数见下表。

> 磁盘IO地址和对应功能

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，需要表明要读写几个扇区。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；第6位：为1=LBA模式，0 = CHS模式（Cylinder/Head/Sector）；第7位和第5位必须为1
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

其中0x1f7的状态寄存器第7位表示控制器是否忙碌，第6位表示磁盘驱动器是否准备好了，在读取硬盘前需要等待硬盘就绪，然后通过设置相关参数，对0x1f7写入读数据的命令后，从0x1f0读取数据。

bootloader关于读取硬盘扇区的函数有两个，分别是readsect与readseg，其中readseg需要调用readsect。

readsect：

readsect函数有两个参数，其中一个为存放读取到数据的位置，另一个为LBA参数，通过设置参数读取一个硬盘扇区的数据到指定位置，具体的实现如下：

```
static void waitdisk(void) {
    while ((inb(0x1F7) & 0xC0) != 0x40)          //等待就绪(0x40=01000000)
}
//dst：将数据读取到哪个位置，secno：LBA参数
static void    readsect(void *dst, uint32_t secno) {
    // 等待就绪
    waitdisk();
    //设置参数
    outb(0x1F2, 1);                              // 读1个扇区
    outb(0x1F3, secno & 0xFF);
    outb(0x1F4, (secno >> 8) & 0xFF);
    outb(0x1F5, (secno >> 16) & 0xFF);
    outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);  // 0xE0=11100000
    outb(0x1F7, 0x20);                           // 读取扇区命令：0x20
    // 等待就绪
    waitdisk();
    // 读取扇区
    insl(0x1F0, dst, SECTSIZE / 4);              // 通过0x1F0端口读取数据至dst（insl：一
次读四个字节）
}
```

readseg：

readseg函数有三个参数，va是一个虚拟内存位置，读取到的数据从va的位置开始存放，count表示读取数据的字节数，offset表示偏移，在开始读取前va减去offset%sectsize，这表明存放数据的虚拟地址也是按照扇区大小对齐的，开始读的扇区号为offset/sectsize+1，表示从偏移offset个字节处所在的扇区开始读，并跳过主引导扇区，如offset=700，512<700<1024，则从第2个扇区开始读。每次读一个扇区的数据并更新va，直到已经读取了超过需要读取的字节数。（可能会多读取一些数据，但没有影响）

具体实现如下：

```
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;           // OS的elf文件从扇区1开始，扇区
0是主引导扇区
    for (; va < end_va; va += SECTSIZE, secno ++) {
        readsect((void *)va, secno);
    }
}
```

2.ELF格式的OS

ELF(Executable and linking format)文件格式是Linux系统下的一种常用目标文件(object file)格式，有三种主要类型，在本实验中使用的类型为用于执行的可执行文件(executable file)，用于提供程序的进程映像，加载的内存执行。在ELF文件的开始处有一个ELF header，用于描述ELF文件的组织。在libs目录下打开elf.h，可以找到ELF header的定义：

```
struct elfhdr {
    uint32_t e_magic;     // must equal ELF_MAGIC
    uint8_t e_elf[12];
    uint16_t e_type;      // 1=relocatable, 2=executable, 3=shared object, 4=core
image
    uint16_t e_machine;   // 3=x86, 4=68K, etc.
    uint32_t e_version;   // file version, always 1
    uint32_t e_entry;     // entry point if executable
    uint32_t e_phoff;     // file position of program header or 0
    uint32_t e_shoff;     // file position of section header or 0
    uint32_t e_flags;     // architecture-specific flags, usually 0
    uint16_t e_ehsize;    // size of this elf header
    uint16_t e_phentsize; // size of an entry in program header
    uint16_t e_phnum;     // number of entries in program header or 0
    uint16_t e_shentsize; // size of an entry in section header
    uint16_t e_shnum;     // number of entries in section header or 0
    uint16_t e_shstrndx;  // section number that contains section name strings
};
```

其中，e_magic成员变量必须等于ELF_MAGIC幻数，用于检验是否为合法的可执行文件。e_entry成员变量指定了该可执行文件程序入口的虚拟地址，而 e_phoff成员变量表示program header表的位置偏移量，可以根据它查找到program header。program header描述与程序执行直接相关的目标文件结构信息，用来在文件中定位各个段的映像，同时包含其他一些用来为程序创建进程映像所必需的信息。bootloader通过ELF Header中的e_phoff在program header表中找到program header，将相应的段，读到内存中来。program header结构如下：

```
struct proghdr {
    uint32_t p_type;   // 段的类型，说明本段为代码段或数据段或其他
    uint32_t p_offset; // 段相对于文件头的偏移
    uint32_t p_va;     // 段的内容被放到内存中的虚拟地址
    uint32_t p_pa;     // physical address, not used
    uint32_t p_filesz; // size of segment in file
    uint32_t p_memsz;  // 段在内存映像中占用的字节数
    uint32_t p_flags;  // read/write/execute bits
    uint32_t p_align;  // required alignment, invariably hardware page size
};
```

3.bootloader加载ELF格式的OS

了解了硬盘扇区的读取和ELF格式的相关内容，就可以分析bootmain.c中bootloader是如何加载ELF格式的OS的了。

进入bootmain中后，首先读取8个扇区的内容（从1号扇区开始），使ELFHDR指针指向ELF文件的头部，即指向elf header。然后通过幻数检验该文件是否是正确的ELF文件，如果是则继续后续的操作。读取扇区后，定义了program header类型的指针ph和eph，并通过e_phoff找到program header的位置，使ph指向program header，根据e_phnum使eph指向program header结束的位置。然后根据program header中的信息，调用readseg函数，通过ph中程序段的大小，相对于文件头的偏移，将各个程序段的内容放入指定的虚拟地址处，完成OS的加载。最后，根据ELFHDR的入口信息，进入并开始执行内核程序。

```
void
bootmain(void) {
    // 读取8个扇区
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);
    // 检验ELF是否有效
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }
    struct proghdr *ph, *eph;
    // 加载程序段
    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);
    }
    // 根据ELFHDR中的入口信息，进入内核程序，启动OS
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();
bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);
    /* do nothing */
    while (1);
}
```

练习5：实现函数调用堆栈跟踪函数

根据要求，需要完成kdebug.c中函数print_stackframe的实现，并输出栈帧和调用相关信息。

函数调用和建立栈帧的过程是从call指令开始的，call指令首先将call指令的下一条指令地址，也就是调用函数的返回地址入栈，然后将ebp入栈，以用于恢复调用者的栈帧，接下来将esp的值赋给ebp，更新帧指针，esp减小以开辟栈空间，完成调用函数的栈帧建立。另外，在使用call指令进行函数调用前，还会将参数入栈。因此在函数调用的过程中，栈帧的情况如下所示：

```
|  ...                              高地址
|  参数2
|  参数1
|  返回地址
|  保存的[ebp]       <-------- ebp 帧指针
|  局部变量等
|  ...
|  ...              <-------- esp 栈指针
```

了解了函数调用时栈帧的建立，可以打开kdebug.c完成堆栈跟踪函数了。在注释中有关于eip，esp和ebp的一些解释及其他函数的说明，并给出了实现print_stackframe的提示，其中重要的注释如下：

```
/*
 * 内联read_ebp() 可以给出当前ebp的值
 * 非内联函数read_eip()可以读取当前eip的值，从堆栈中获得调用者的eip
 * 在print_debuginfo()函数中，函数debuginfo_eip()可以获取关于调用链足够的信息。最终
print_stackframe()函数会最终并打印这些信息。
 */
void
print_stackframe(void) {
     /* LAB1 YOUR CODE : STEP 1 */
     /* (1) 调用read_ebp()得到ebp的值。类型为(uint32_t);
      * (2) 调用read_eip()得到eip的值。类型为(uint32_t);
      * (3) 从0到STACKFRAME_DEPTH（20）
      *    (3.1) 打印ebp，eip的值
      *    (3.2) (uint32_t) 调用参数[0..4] = 地址的内容(uint32_t)ebp +2 [0..4]
      *    (3.3) cprintf("\n");
      *    (3.4) 调用print_debuginfo(eip-1)打印函数名和行号等信息。
      *    (3.5) 弹出调用栈帧
      *           注意: 调用函数的返回地址eip  = ss:[ebp+4]
      *                 调用函数的ebp = ss:[ebp]
      */
}
```

根据给出的提示输出ebp，eip的值，并输出函数的参数，参数共打印4个，参数开始的位置为ebp+8（指针运算时为ebp+2），并调用print_debuginfo打印函数名和行号等信息，最后更新eip和ebp，找到调用者的栈帧并继续打印信息。并且需要注意ebp存在边界0。其中ebp和eip的值为32位无符号整型，作指针使用需要进行类型转换。具体实现如下：

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

make debug得到了要求实现的以下输出：

```
...
ebp:00007b28 eip:00100ab3 args:0x00010094 0x00010094 0x00007b58 0x00100096
    kern/debug/kdebug.c:306: print_stackframe+25
ebp:00007b38 eip:00100dac args:0x00000000 0x00000000 0x00000000 0x00007ba8
    kern/debug/kmonitor.c:125: mon_backtrace+14
ebp:00007b58 eip:00100096 args:0x00000000 0x00007b80 0xffff0000 0x00007b84
    kern/init/init.c:48: grade_backtrace2+37
ebp:00007b78 eip:001000c4 args:0x00000000 0xffff0000 0x00007ba4 0x00000029
    kern/init/init.c:53: grade_backtrace1+42
ebp:00007b98 eip:001000e7 args:0x00000000 0x00100000 0xffff0000 0x0000001d
    kern/init/init.c:58: grade_backtrace0+27
ebp:00007bb8 eip:00100111 args:0x0010343c 0x00103420 0x0000130a 0x00000000
    kern/init/init.c:63: grade_backtrace+38
ebp:00007be8 eip:00100055 args:0x00000000 0x00000000 0x00000000 0x00007c4f
    kern/init/init.c:28: kern_init+84
ebp:00007bf8 eip:00007d74 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
    <unknow>: -- 0x00007d73 --
++ setup timer interrupts
...
```

最后一行结束后表示更新的ebp为0，即已经找到了第一个使用栈的函数。bootloader在完成实模式到保护模式的转换后建立了栈空间，将ebp设置为0，call bootmian进入bootmain函数，这就是第一次函数调用。故最后一行的ebp：0x7bf8为bootmain的帧指针，eip：0x7d74为bootmain中进入OS处。bootblock.asm中可以找到此处。

```
    // call the entry point from the ELF header
    // note: does not return
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();
    7d67:    a1 ec 7d 00 00           mov    0x7dec,%eax
    7d6c:    8b 40 18                 mov    0x18(%eax),%eax
    7d6f:    25 ff ff ff 00           and    $0xffffff,%eax
    7d74:    ff d0                    call   *%eax
```

在调用bootmain时没有传入参数，而ebp+8的位置为0x7c00，是bootloader代码开始的地方，故args是bootloader开始处的指令机器码，可以打开bootblock.asm，前三条指令对应机器码为fa，fc，31，c0，小端法下读出为0xc031fcfa，与args打印的第一个参数对应验证。

```
.code16                                             # Assemble for 16-bit mode
    cli                                             # Disable interrupts
    7c00:    fa                      cli
    cld                                             # String operations increment
    7c01:    fc                      cld
    xorw %ax, %ax                                   # Segment number zero
    7c02:    31 c0                   xor    %eax,%eax
...
```

练习6：完善中断初始化和处理

1.中断与中断描述符表

操作系统的中断有三种：由时钟或其他外设引起的**中断**（外部中断），由程序执行非法指令等引起的**异常**（内部中断），由程序请求系统服务的**系统调用**（陷阱中断，软中断）。而操作系统需要实现中断处理机制，对发生的中断进行处理。在保护模式下，中断是这样处理的：当CPU收到中断信号（由中断控制器发出）后，会暂停执行当前的程序，跳转到处理该中断的例程中，完成中断处理后再继续执行被打断的程序。其中，中断服务例程的信息被存放在中断描述符表（IDT）中，而中断描述符表的起始地址保存在IDTR寄存器中，CPU收到中断信号后会读取一个中断向量，以中断向量为索引，就可以在IDT表中找到相应的中断描述符，从而获得中断服务例程的起始地址并执行中断服务例程。

中断描述符表中的表项被称为**门描述符**，每个门描述符为8个字节。在kern/mm/mmu.h中可以找到门描述符的定义。可以看到，在8个字节共64位中，0-15位为偏移的低16位，48-63位为偏移的高16位，16-31位为段选择子。

```
struct gatedesc {
    unsigned gd_off_15_0 : 16;        // low 16 bits of offset in segment
    unsigned gd_ss : 16;              // segment selector
    unsigned gd_args : 5;             // # args, 0 for interrupt/trap gates
    unsigned gd_rsv1 : 3;             // reserved(should be zero I guess)
    unsigned gd_type : 4;             // type(STS_{TG,IG32,TG32})
    unsigned gd_s : 1;                // must be 0 (system)
    unsigned gd_dpl : 2;              // descriptor(meaning new) privilege level
    unsigned gd_p : 1;                // Present
    unsigned gd_off_31_16 : 16;       // high bits of offset in segment
};
```

根据段选择子可以在GDT表中找到段描述符，得到段的起始地址，结合段的偏移，就可以得到中断服务例程的起始地址。

2.中断向量表初始化

中断向量表的初始化需要通过idt_init()函数完成，该函数在trap.c中，需要完成其具体实现，在注释中给出了提示如下：

```
void
idt_init(void) {
     /* LAB1 YOUR CODE : STEP 2 */
     /* (1) 每个中断服务例程的起始地址被存放在__vectors数组中，这个数组是由vector.c生成，保存在vector.S当中。
      *     可以使用extern uintptr_t __vectors[];引入并使用这个数组。
      * (2) 接下来可以设置IDT中的表项了，idt[256]就是中断描述符表，可以使用SETGATE宏定义设置IDT表中的每一项。
      * (3) 设置好中断描述符表后，只需要执行lidt指令，cpu就可以找到中断描述符表了。（lidt将IDT表的起始位置和界限放入IDTR寄存器）
      */
}
```

设置IDT表中的门描述符通过SETGATE宏实现，SETGATE类似函数，有五个参数。五个参数的意义如下：

- gate：需要赋值设置的门描述符，即此处传入idt中的元素进行设置
- istrap：设置门描述符的的类型，此处无论传入什么值都将门描述符类型设置为陷阱门描述符
- sel：段选择子
- off：偏移

- dpl：描述符所指向中断服务例程的特权级

```
#define SETGATE(gate, istrap, sel, off, dpl) {
    (gate).gd_off_15_0 = (uint32_t)(off) & 0xffff;          //偏移
    (gate).gd_ss = (sel);                                   //段选择子
    (gate).gd_args = 0;
    (gate).gd_rsv1 = 0;
    (gate).gd_type = (istrap) ? STS_TG32 : STS_IG32;        //描述符类型设置
    (gate).gd_s = 0;
    (gate).gd_dpl = (dpl);                                  //特权级
    (gate).gd_p = 1;
    (gate).gd_off_31_16 = (uint32_t)(off) >> 16;            //偏移
}
```

接下来考虑传入的参数，第一个参数直接传入需要设置的idt[i]，第二个参数传入0或1都可以，第三个参数为段选择子，中断服务例程由操作系统执行，因此应该位于操作系统内核的代码段，在memlayout.h文件中有各个段的信息如下：

```
/* global descriptor numbers */
#define GD_KTEXT    ((SEG_KTEXT) << 3)        // kernel text
#define GD_KDATA    ((SEG_KDATA) << 3)        // kernel data
#define GD_UTEXT    ((SEG_UTEXT) << 3)        // user text
#define GD_UDATA    ((SEG_UDATA) << 3)        // user data
#define GD_TSS        ((SEG_TSS) << 3)        // task segment selector
```

故第三个参数传入内核代码段的选择子，即GD_KTEXT。第四个参数为偏移，这个参数应该是由vector提供的，只需要传入对应的vector的元素。最后一个参数为特权级（**Descriptor Privilege Level，描述符特权级**，规定访问该段的权限级别），由于中断服务例程在内核态运行，应设置为内核特权级，在memlayout.h中定义为DPL_KERNEL(0)，需要注意系统调用是在用户态下进行的，系统调用的DPL需要设置为用户特权级，在memlayout.h中定义为DPL_USER (3)，系统调用会将特权级切换为内核特权级，在trap.h中可以找到其中断号。

```
//memlayout.h中的DPL定义
#define DPL_KERNEL    (0)
#define DPL_USER      (3)
//trap.h中定义的特权级切换中断号
#define T_SWITCH_TOU                120    // user/kernel switch
#define T_SWITCH_TOK                121    // user/kernel switch
```

初始化中断描述符表后还需要使用lidt指令将idt的起始地址和界限装载入IDTR寄存器，该函数在x86.h中定义，起始地址和界限在trap.c中定义的idt_pd中。

```
static inline void
lidt(struct pseudodesc *pd) {
    asm volatile ("lidt (%0)" :: "r" (pd));
}
static struct pseudodesc idt_pd = {
    sizeof(idt) - 1, (uintptr_t)idt
};
```

综上，可实现idt_init()函数如下：

```
void
idt_init(void) {
    extern uintptr_t __vectors[];
    int num=sizeof(idt)/sizeof(struct gatedesc);
    for(int i=0;i<num;i++){
        SETGATE(idt[i],1,GD_KTEXT,__vectors[i],DPL_KERNEL);
    }
    SETGATE(idt[T_SWITCH_TOK],GD_KTEXT,__vectors[T_SWITCH_TOK],DPL_USER);
    lidt(&idt_pd);
}
```

3.中断处理

实验要求完成中断处理函数trap，trap函数调用了trap_dispatch()，这个函数会根据trap类型进行中断处理。

```
void
trap(struct trapframe *tf) {
    // dispatch based on what type of trap occurred
    trap_dispatch(tf);
}
```

其中的时钟中断处理部分，只需要使用全局变量对时钟中断进行计数，当时钟中断100次,即TICK_NUM次后，调用print_ticks()打印“100 ticks”，补充后的trap_dispatch()如下:

```
static void
trap_dispatch(struct trapframe *tf) {
    char c;

    switch (tf->tf_trapno) {
    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!
         */
        ticks++;
        if(ticks%TICK_NUM==0) print_ticks();
        break;
    case IRQ_OFFSET + IRQ_COM1:
        c = cons_getc();
        cprintf("serial [%03d] %c\n", c, c);
        break;
    case IRQ_OFFSET + IRQ_KBD:
        c = cons_getc();
        cprintf("kbd [%03d] %c\n", c, c);
        break;
    //LAB1 CHALLENGE 1 : YOUR CODE you should modify below codes.
    case T_SWITCH_TOU:
    case T_SWITCH_TOK:
        panic("T_SWITCH_** ??\n");
        break;
    case IRQ_OFFSET + IRQ_IDE1:
    case IRQ_OFFSET + IRQ_IDE2:
        /* do nothing */
        break;
    default:
        // in kernel, it must be a mistake
        if ((tf->tf_cs & 3) == 0) {
            print_trapframe(tf);
            panic("unexpected trap in kernel.\n");
        }
    }
}
```

完成后执行make debug命令，可以看到打印出的”100 ticks“。

```
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
...
```

扩展练习一

根据实验要求，需要完成kern_init.c中的lab1_switch_test()，该函数调用了lab1_switch_to_kernel() 和lab1_switch_to_user()两个函数，这两个函数分别实现了用户态到内核态的切换及内核态到用户态的切换，该函数如下：

```
static void
lab1_switch_test(void) {
    lab1_print_cur_status();
    cprintf("+++ switch to  user  mode +++\n");
    lab1_switch_to_user();
    lab1_print_cur_status();
    cprintf("+++ switch to kernel mode +++\n");
    lab1_switch_to_kernel();
    lab1_print_cur_status();
}
```

为了实现特权级的切换，需要了解以下内容：

特权级

特权级：操作系统通过特权级判断是否可访问数据。在ucore中只有两个特权级，0为内核态，3为用户态。有以下三种不同的特权级：

- CPL：当前特权级，表示当前执行的代码的特权级，保存在CS或SS寄存器的特定位中，执行不同的代码时，CPL也会随之改变，表示特权级发生变化。
- DPL：描述符特权级，表示访问该段的最低特权级，特权级必须高于DPL才可以访问。保存在段描述符中。
- RPL：请求特权级，表示当前代码段发出了一个特定特权级的请求，判断能否访问时，选择 CPL 与 RPL 较大的值（即低的特权级）来判定是否有权进行访问。

中断处理

ucore中发生中断后，处理的过程如下：

首先，中断发生后，以中断向量为索引，在中断描述符表IDT中获得中断描述符，根据中断描述符的段选择子找到GDT中的段描述符，通过段基址和段偏移量，找到中断服务例程的位置，并跳转执行中断服务例程。

在vector.S可以看到进入中断服务例程后，首先将中断号压栈，然后跳转进入__alltraps进行处理。在__alltraps中首先将一些寄存器压栈，并调用trap函数。向trap函数传入的参数为trapframe类型的指针，trapframe结构中保存的是进入中断前的信息，在中断处理结束后，需要利用这些信息恢复进入中断前的状态。

```
//进入中断服务例程
vector0:
  pushl $0
  pushl $0
  jmp __alltraps
//trapentry.s: __alltraps
.globl __alltraps
__alltraps:
    # push registers to build a trap frame
    # therefore make the stack look like a struct trapframe
    pushl %ds
    pushl %es
    pushl %fs
    pushl %gs
    pushal
    # load GD_KDATA into %ds and %es to set up data segments for kernel
    movl $GD_KDATA, %eax
    movw %ax, %ds
    movw %ax, %es
    # push %esp to pass a pointer to the trapframe as an argument to trap()
    pushl %esp
    # call trap(tf), where tf=%esp
    call trap
    # pop the pushed stack pointer
    popl %esp
    # return falls through to trapret...
```

trapframe的定义在trap.h中，其中的tf_esp和tf_ss是在发生特权级切换时需要保存的信息，如下：

```
struct trapframe {
    struct pushregs tf_regs;
    uint16_t tf_gs;
    uint16_t tf_padding0;
    uint16_t tf_fs;
    uint16_t tf_padding1;
    uint16_t tf_es;
    uint16_t tf_padding2;
    uint16_t tf_ds;
    uint16_t tf_padding3;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding4;
    uint32_t tf_eflags;
    /* below here only when crossing rings, such as from user to kernel */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding5;
} __attribute__((packed));
```

需要注意的是**中断发生时的压栈**。esp寄存器和ss寄存器的值只有在发生特权级转换时才需要压栈，如从用户特权级转向内核特权级，会将当前栈由用户栈转为内核栈，需要保存esp和ss寄存器用于恢复用户栈，返回时执行iret指令，将trapframe中的信息恢复，发现栈中弹出的cs寄存器值所标记的特权级（原来的特权级）和当前cs显示的特权级不同，说明特权级发生了转换，就会对esp，ss寄存器弹栈，恢复原来的栈。不发生特权级转换时不需要进行栈的切换，这两个寄存器不会被压栈。

特权级切换的实现

本练习要求的特权级切换，是通过产生中断的方式实现的。在lab1_switch_to_kernel() 和lab1_switch_to_user()两个函数中产生中断，进入中断处理例程，中断处理结束后将从trapframe恢复原状态，将trapframe中的寄存器信息进行修改，等到中断结束恢复状态的时候，就实现了状态的转换。

用函数实现内核态转为用户态时，进入中断时特权级没有发生变化，因此esp和ss没有压栈，而为了实现特权级转换，修改了trapframe的值，在弹栈时会弹出esp和ss切换栈，因此需要留出空间，并在中断处理时将值修改为用户栈的值，使中断完成后能实现栈的切换。另外，实际上es段，ss段在ucore中都是DS段，没有区别，具体实现如下：

```
//
static void
lab1_switch_to_user(void) {
    //LAB1 CHALLENGE 1 : TO DO
    asm volatile(
    "sub $0x8,%%esp \n"            //留出ss，esp的空间
    "int %0 \n"                    //中断
    "movl %%ebp,%%esp"             //恢复栈指针
    :
    :"i"(T_SWITCH_TOU)             //中断号
    );
}
//trap_dispatch中处理中断
case T_SWITCH_TOU:
    tf->tf_cs=USER_CS;
    tf->tf_ds=USER_DS;
    tf->tf_es=USER_DS;
    tf->tf_ss=USER_DS;
    tf-tf_eflags|= FL_IOPL_MASK;  //根据答案，此处设置的flag是为了用户能正常进行IO操作
    break;
```

用函数实现用户态转为内核态时，由于中断时已经发生了特权级的变化，不需要预留空间，只需要修改trapframe。另外，由于将cs的值修改为内核态下的值（等于当前cs的值），cpu认为没有发生特权级转换，不会发生ss和esp的弹栈去恢复栈空间。

```
tatic void
lab1_switch_to_kernel(void) {
 /                /LAB1 CHALLENGE 1 : TODO
asm volatile (
     "int %0 \n"
     "movl %%ebp, %%esp \n"
     :
     : "i"(T_SWITCH_TOK)
     );
}
//trap_dispatch中处理中断
case T_SWITCH_TOK:
     tf->tf_cs = KERNEL_CS;
     tf->tf_ds = KERNEL_DS;
     tf->tf_es = KERNEL_DS;
     break;
```

实验总结

与参考答案的对比

练习一

与参考答案进行分析的顺序不同，先从ucore.img的生成开始分析，再分析bootblock和kernel的生成，比参考答案有更多参数说明。

练习二

直接输入命令make lab1-mon，使用lab1init中的命令进行调试，没有将运行的汇编指令保存到log文件中。

练习三

补充了关于实模式与保护模式，分段机制与全局描述符表，A20的相关知识。具体分析过程与参考答案基本一致。

练习四

更加详细介绍了关于硬盘扇区读取，elf文件格式的相关知识，对于加载elf格式的OS的过程比参考答案有更详尽的解释。

练习五

具体实现基本一致。比参考答案多补充了函数调用时栈帧的建立，以及最后一行输出中参数的解释与验证。

练习六

比答案多补充了中断描述符表，中断机制的相关知识。函数的实现与答案基本一致，有完整的分析过程和更详细的解释。

扩展练习一

主要根据参考网上的实现方法，学习和理解特权级切换的过程和实现。答案中重新设置了一个trapframe，让中断结束后从这个trapframe恢复信息，实现切换。关于一些实现特权级切换的细节和内容仍然不够清楚，有待后续继续学习。

重要知识点

- Makefile的格式
- 操作系统镜像文件的构成

- CPU加电后的第一条指令及BIOS完成的工作
- gdb调试方法
- 实模式与保护模式
- 分段机制
- 全局描述符表及其初始化
- 硬盘扇区读取
- ELF文件格式
- 中断机制及中断描述符表

涉及的OS原理

- 地址空间
- 地址转换机制
- 分段机制
- 中断机制

本实验主要是关于操作系统启动的过程的具体实现，地址空间管理使用分段机制，，并涉及全局描述符表，中断描述符表等操作系统启动时初始化的具体实现。

未涉及的内容

- 进程及进程调度
- 分页机制
- 空闲空间管理