练习1

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)

ucore.img makefile

```
# -------------------------------------------------------------------

# 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)

# >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

首先利用dd命令开辟10000字节空间

然后依次将bootblock和kernel的elf放入空间中

该指令需要先生成bootblock和kernel两个可执行文件

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

生成bootblock

```
# 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 $(call toobj,bootblock)
    @$(OBJDUMP) -S $(call objfile,bootblock) > $(call asmfile,bootblock)
    @$(OBJCOPY) -S -O binary $(call objfile,bootblock) $(call outfile,bootblock)
```

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

$(call create_target,bootblock)

# -------------------------------------------------------------------
```

首先编译源文件生成o文件

然后使用ld链接生成bootblock

可以在输出命令中看到

```
+ cc Sbootasm.S
gcc -Iboot/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-protector -Ilibs/
+ cc boot/bootmain.c
gcc -Iboot/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-protector -Ilibs/
+ 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.
```

生成kernel

首先编译源文件生成o文件

然后使用ld链接生成kernel

```
gcc -Ikern/driver/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gst
+ cc kern/driver/picirq.c
gcc -Ikern/driver/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-protector
+ cc kern/trap/trap.c
gcc -Ikern/trap/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-protector -I
+ cc kern/trap/trapentry.S
gcc -Ikern/trap/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-protector -I
+ cc kern/trap/vectors.S
gcc -Ikern/trap/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-protector -I
+ cc kern/mm/pmm.c
gcc -Ikern/mm/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-protector -Il
+ cc libs/printfmt.c
gcc -Ilibs/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-protector -Ilibs/
+ cc libs/string.c
gcc -Ilibs/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-protector -Ilibs/
+ ld bin/kernel
ld -m    elf_i386 -nostdlib -T tools/kernel.ld -o bin/kernel obj/kern/init/init.o obj/kern/libs/readline.o ob
```

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

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?

```c
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]);
```

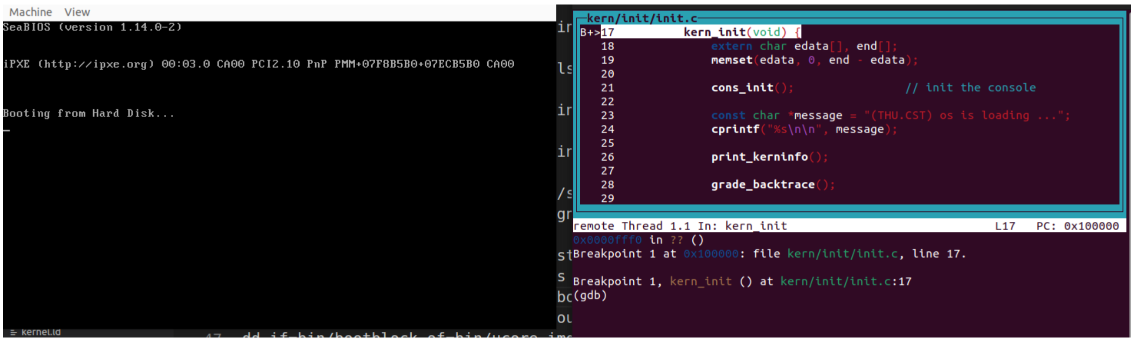
长度为512字节

结尾两个字节是0x55 0xAA

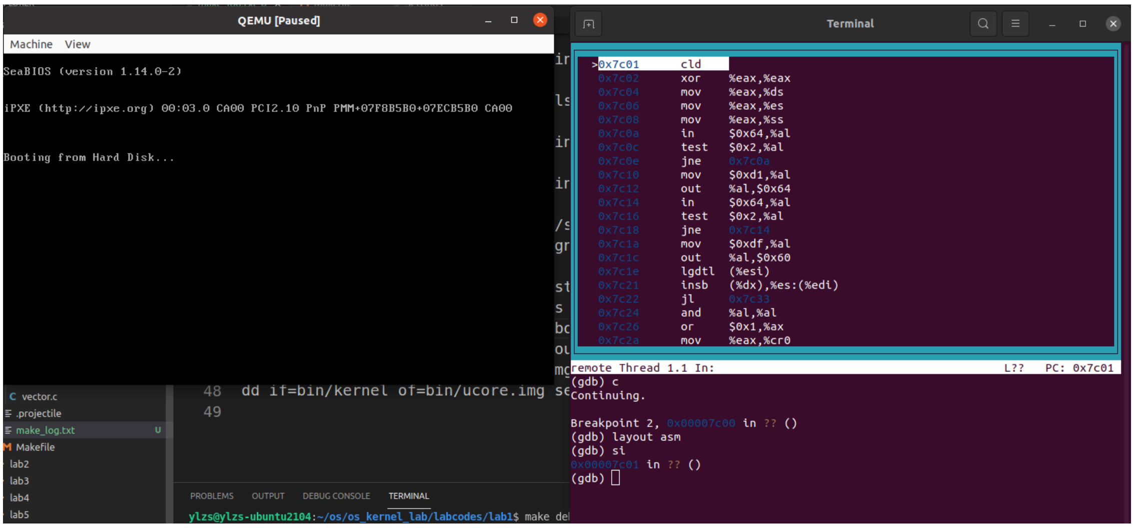
练习2

使用qemu执行并调试lab1中的软件。（要求在报告中简要写出练习过程）

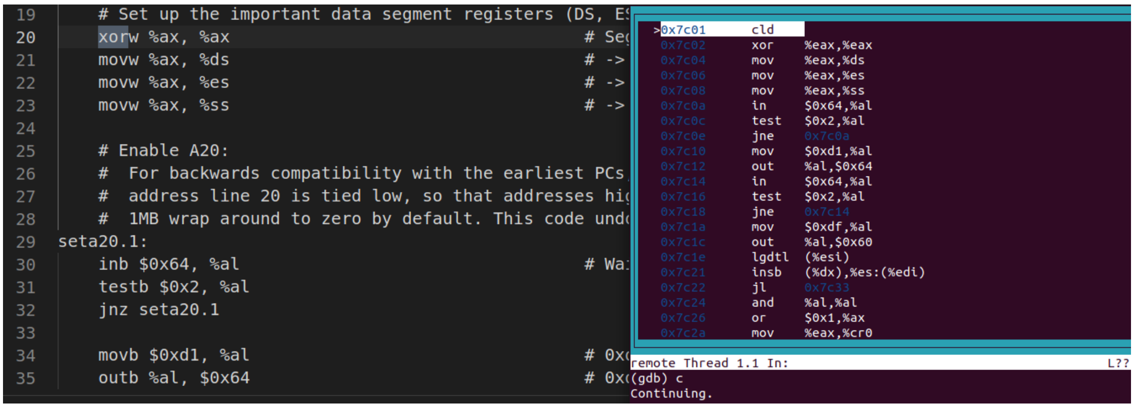
1. 从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行。



2. 在初始化位置0x7c00设置实地址断点,测试断点正常。

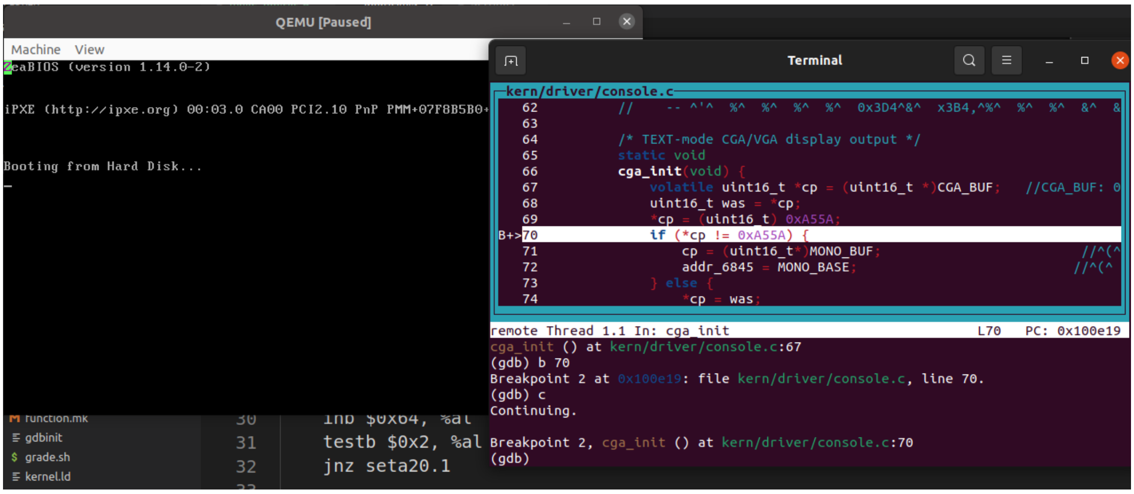


3. 从0x7c00开始跟踪代码运行,将单步跟踪反汇编得到的代码与bootasm.S和 bootblock.asm进行比较。



可以看到反汇编的代码和bootblock.asm的代码一致

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



练习3

分析bootloader进入保护模式的过程。 （要求在报告中写出分析）

关闭中断，将各个段寄存器重置

```
.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
```

首先禁止中断

将ax置零

接着将ds es ss寄存器都置零

开启A20

```
# 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
```

开启A20地址线之后，用来表示内存地址的位数变多了。

开启前20位，开启后是32位。

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

进入保护模式

初始化GDT表：从引导区加载GDT表

```
lgdt gdtdesc
```

进入保护模式：将cr0寄存器PE位置1

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

通过长跳转更新cs的基地址

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

设置段寄存器，栈指针

```
.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
```

转到保护模式完成，进入boot主方法

```
 call bootmain
```

练习4

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

```
/* 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) {
        goto bad;
    }

    struct proghdr *ph, *eph;

    // load each program segment (ignores ph flags)
    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);
    }

    // call the entry point from the ELF header
    // note: does not return
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();

bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);

    /* do nothing */
    while (1);
}
```

结合bootmain.c可以看到

在bootmain方法中 首先调用readseg读取磁盘的第一页

```
/* *
 * readseg - read @count bytes at @offset from kernel into virtual address @va,
 * might copy more than asked.
 * */
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
    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);
    }
}
```

查看readseg函数 可以看到该函数读取 起始地址为offset长度为count的数据复制到首地址为va的空间中

然后bootmain检查ELF文件是否符合规范 如果不符合规范 则直接使用goto语句跳转到boot失败

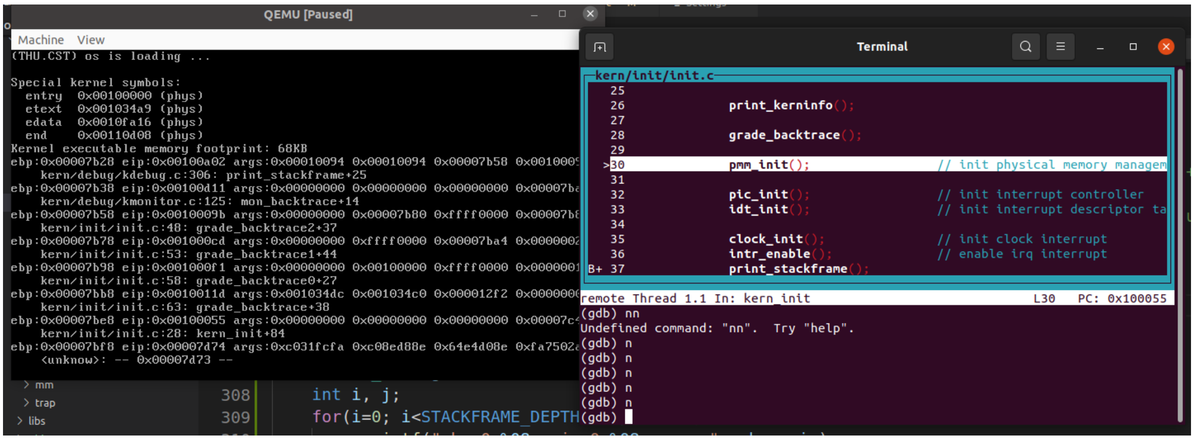
如果ELF文件符合规范 则继续加载ELF文件的程序段

最后通过call ELF的e_entry将控制权转给OS

练习5

实现函数调用堆栈跟踪函数 （需要编程）

```
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();
    uint32_t eip = read_eip();
    uint32_t *arguments;
    int i, j;
    for(i=0; i<STACKFRAME_DEPTH&&ebp; i++){
        cprintf("ebp:0x%08x eip:0x%08x args:", ebp, eip);
        arguments = (uint32_t*)ebp + 2; // ebp + 8
        for(j=0;j<4;j++){
            cprintf("0x%08x ", arguments[j]);
        }
        cprintf("\n");
        print_debuginfo(eip-1);
        eip = ((uint32_t*)ebp)[1];// ebp + 4 return address
        ebp = ((uint32_t*)ebp)[0];// ebp + 0 old ebp
    }
}
```



练习6

完善中断初始化和处理 （需要编程）

编写idt_init函数初始化中断描述表

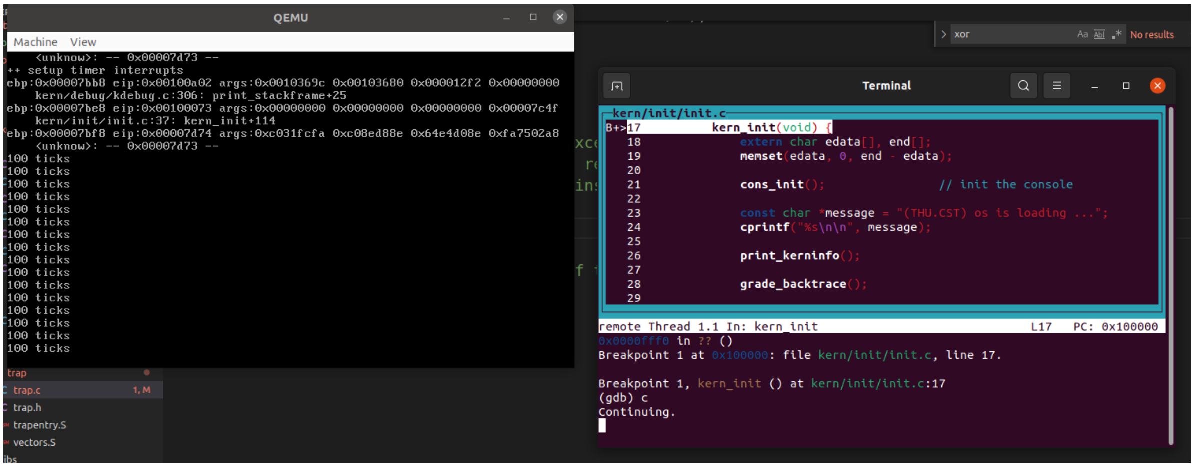
```
/* idt_init - initialize IDT to each of the entry points in kern/trap/vectors.S
*/
void
idt_init(void) {
     /* LAB1 YOUR CODE : STEP 2 */
     /* (1) Where are the entry addrs of each Interrupt Service Routine (ISR)?
      *     All ISR's entry addrs are stored in __vectors. where is uintptr_t
__vectors[] ?
      *     __vectors[] is in kern/trap/vector.S which is produced by
tools/vector.c
      *     (try "make" command in lab1, then you will find vector.S in
kern/trap DIR)
      *     You can use  "extern uintptr_t __vectors[];" to define this extern
variable which will be used later.
      * (2) Now you should setup the entries of ISR in Interrupt Description
Table (IDT).
      *     Can you see idt[256] in this file? Yes, it's IDT! you can use
SETGATE macro to setup each item of IDT
      * (3) After setup the contents of IDT, you will let CPU know where is the
IDT by using 'lidt' instruction.
      *     You don't know the meaning of this instruction? just google it! and
check the libs/x86.h to know more.
      *     Notice: the argument of lidt is idt_pd. try to find it!
      */
     extern uintptr_t __vectors[];
     int i=0;
     // init idt
     for(; i < 256; i++){
         /* *
         * Set up a normal interrupt/trap gate descriptor
         *   - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate
         *   - sel: Code segment selector for interrupt/trap handler
         *   - off: Offset in code segment for interrupt/trap handler
         *   - dpl: Descriptor Privilege Level - the privilege level required
         *          for software to invoke this interrupt/trap gate explicitly
         *          using an int instruction.
         * */
         SETGATE(idt[i], 0, KERNEL_CS, __vectors[i], DPL_KERNEL);
     }
     SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, __vectors[T_SWITCH_TOK], DPL_USER);
     // set idtr register
     lidt(&idt_pd);
}
```

完成中断类型的判别并利用中断打印ticks

```
/* trap_dispatch - dispatch based on what type of trap occurred */
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");
        }
    }
}
```



扩展练习

challenge1

修改init.c的用户态和内核态切换函数如下

```
static void
lab1_switch_to_user(void) {
    //LAB1 CHALLENGE 1 : TODO
    asm volatile ("int %0\n"::"i"(T_SWITCH_TOU));
}

static void
lab1_switch_to_kernel(void) {
    //LAB1 CHALLENGE 1 :  TODO
    asm volatile ("int %0\n"::"i"(T_SWITCH_TOK));
}
```

在trap.c中对模式切换的中断进行判断和处理

```
* trap_dispatch - dispatch based on what type of trap occurred */
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:
        if(tf->tf_cs != USER_CS){
            switchk2u = *tf;
            switchk2u.tf_cs = USER_CS;
            switchk2u.tf_ds = switchk2u.tf_es = switchk2u.tf_ss = USER_DS;
            switchk2u.tf_esp = (uint32_t)tf + sizeof(struct trapframe) - 8;
            switchk2u.tf_eflags |= FL_IOPL_MASK;
```

```
            *((uint32_t*)tf - 1) = (uint32_t)&switchk2u;
        }
        break;
    case T_SWITCH_TOK:
        // panic("T_SWITCH_** ??\n");
        if (tf->tf_cs != KERNEL_CS) {
            tf->tf_cs = KERNEL_CS;
            tf->tf_ds = tf->tf_es = KERNEL_DS;
            tf->tf_eflags &= ~FL_IOPL_MASK;
            switchu2k = (struct trapframe *)(tf->tf_esp - (sizeof(struct
trapframe) - 8));
            memmove(switchu2k, tf, sizeof(struct trapframe) - 8);
            *((uint32_t *)tf - 1) = (uint32_t)switchu2k;
        }
        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");
        }
    }
}
```

测试结果

```
ylzs@ylzs-ubuntu2104:~/os/os_kernel_lab/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
ylzs@ylzs-ubuntu2104:~/os/os_kernel_lab/labcodes/lab1$
```

扩展练习2

进一步修改trap.c 实现对键盘事件的处理

按0 切换至用户态

按3 切换至内核态

```
/* tmp trapframe record status */
struct trapframe switchk2u, *switchu2k;
/* trap_dispatch - dispatch based on what type of trap occurred */
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);
        if(c == '0'){
            switchk2u = *tf;
            switchk2u.tf_cs = USER_CS;
            switchk2u.tf_ds = switchk2u.tf_es = switchk2u.tf_ss = USER_DS;
            switchk2u.tf_esp = (uint32_t)tf + sizeof(struct trapframe) - 8;
            switchk2u.tf_eflags |= FL_IOPL_MASK;
            *((uint32_t*)tf - 1) = (uint32_t)&switchk2u;
            cprintf("switch to USER MODE!!!\n");
        }else if(c == '3'){
            tf->tf_cs = KERNEL_CS;
            tf->tf_ds = tf->tf_es = KERNEL_DS;
            tf->tf_eflags &= ~FL_IOPL_MASK;
            switchu2k = (struct trapframe *)(tf->tf_esp - (sizeof(struct
trapframe) - 8));
            memmove(switchu2k, tf, sizeof(struct trapframe) - 8);
            *((uint32_t *)tf - 1) = (uint32_t)switchu2k;
            cprintf("switch to KERNEL MODE!!!\n");
        }
        break;
    //LAB1 CHALLENGE 1 : YOUR CODE you should modify below codes.
    case T_SWITCH_TOU:
        if(tf->tf_cs != USER_CS){
            switchk2u = *tf;
            switchk2u.tf_cs = USER_CS;
            switchk2u.tf_ds = switchk2u.tf_es = switchk2u.tf_ss = USER_DS;
            switchk2u.tf_esp = (uint32_t)tf + sizeof(struct trapframe) - 8;

            switchk2u.tf_eflags |= FL_IOPL_MASK;

            *((uint32_t*)tf - 1) = (uint32_t)&switchk2u;
        }
        break;
    case T_SWITCH_TOK:
        // panic("T_SWITCH_** ??\n");
        if (tf->tf_cs != KERNEL_CS) {
            tf->tf_cs = KERNEL_CS;
            tf->tf_ds = tf->tf_es = KERNEL_DS;
            tf->tf_eflags &= ~FL_IOPL_MASK;
```

```
                switchu2k = (struct trapframe *)(tf->tf_esp - (sizeof(struct
trapframe) - 8));
                memmove(switchu2k, tf, sizeof(struct trapframe) - 8);
                *((uint32_t *)tf - 1) = (uint32_t)switchu2k;
            }
            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");
            }
    }
}
```