Lab1

lx1

以下是执行`make "V="`的输出信息

```
user@user-VirtualBox:~/ucore/ucore_os_lab/labcodes/lab1$ make "V="
+ 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
kern/init/init.c:95:1: warning: ‘lab1_switch_test’ defined but not used [-
Wunused-function]
 lab1_switch_test(void) {
 ^
+ cc kern/libs/readline.c
gcc -Ikern/libs/ -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/libs/readline.c -o obj/kern/libs/readline.o
+ cc kern/libs/stdio.c
gcc -Ikern/libs/ -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/libs/stdio.c -o obj/kern/libs/stdio.o
+ cc kern/debug/kdebug.c
gcc -Ikern/debug/ -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/debug/kdebug.c -o obj/kern/debug/kdebug.o
kern/debug/kdebug.c:251:1: warning: ‘read_eip’ defined but not used [-Wunused-
function]
 read_eip(void) {
 ^
+ cc kern/debug/kmonitor.c
gcc -Ikern/debug/ -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/debug/kmonitor.c -o obj/kern/debug/kmonitor.o
+ cc kern/debug/panic.c
gcc -Ikern/debug/ -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/debug/panic.c -o obj/kern/debug/panic.o
kern/debug/panic.c: In function ‘__panic’:
kern/debug/panic.c:27:5: warning: implicit declaration of function
‘print_stackframe’ [-Wimplicit-function-declaration]
     print_stackframe();
     ^
+ cc kern/driver/clock.c
gcc -Ikern/driver/ -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/driver/clock.c -o obj/kern/driver/clock.o
+ cc kern/driver/console.c
gcc -Ikern/driver/ -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/driver/console.c -o obj/kern/driver/console.o
+ cc kern/driver/intr.c
```

```
gcc -Ikern/driver/ -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/driver/intr.c -o obj/kern/driver/intr.o
+ cc kern/driver/picirq.c
gcc -Ikern/driver/ -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/driver/picirq.c -o obj/kern/driver/picirq.o
+ cc kern/trap/trap.c
gcc -Ikern/trap/ -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/trap/trap.c -o obj/kern/trap/trap.o
kern/trap/trap.c:14:13: warning: ‘print_ticks’ defined but not used [-Wunused-
function]
 static void print_ticks() {
             ^
kern/trap/trap.c:30:26: warning: ‘idt_pd’ defined but not used [-Wunused-
variable]
 static struct pseudodesc idt_pd = {
                          ^
+ cc kern/trap/trapentry.S
gcc -Ikern/trap/ -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/trap/trapentry.S -o obj/kern/trap/trapentry.o
+ cc kern/trap/vectors.S
gcc -Ikern/trap/ -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/trap/vectors.S -o obj/kern/trap/vectors.o
+ cc kern/mm/pmm.c
gcc -Ikern/mm/ -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/mm/pmm.c -o obj/kern/mm/pmm.o
+ cc libs/printfmt.c
gcc -Ilibs/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc
 -fno-stack-protector -Ilibs/  -c libs/printfmt.c -o obj/libs/printfmt.o
+ cc libs/string.c
gcc -Ilibs/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc
 -fno-stack-protector -Ilibs/  -c libs/string.c -o obj/libs/string.o
+ 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 obj/kern/libs/stdio.o
obj/kern/debug/kdebug.o obj/kern/debug/kmonitor.o obj/kern/debug/panic.o
obj/kern/driver/clock.o obj/kern/driver/console.o obj/kern/driver/intr.o
obj/kern/driver/picirq.o obj/kern/trap/trap.o obj/kern/trap/trapentry.o
obj/kern/trap/vectors.o obj/kern/mm/pmm.o  obj/libs/printfmt.o obj/libs/string.o
+ 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: 472 bytes
build 512 bytes boot sector: 'bin/bootblock' success!
dd if=/dev/zero of=bin/ucore.img count=10000
记录了10000+0 的读入
记录了10000+0 的写出
5120000字节(5.1 MB)已复制，0.0338003 秒，151 MB/秒
dd if=bin/bootblock of=bin/ucore.img conv=notrunc
记录了1+0 的读入
记录了1+0 的写出
512字节(512 B)已复制，0.00121619 秒，421 kB/秒
dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
记录了146+1 的读入
记录了146+1 的写出
74879字节(75 kB)已复制，0.00106863 秒，70.1 MB/秒
user@user-VirtualBox:~/ucore/ucore_os_lab/labcodes/lab1$
```

分开描述，前面绝大部分都是在运行已经定义好的命令

从最后10行左右看，从bootblock，kernel两个文件中读出数据并写入到ucore.img,

/dev/zero是一个虚拟设备，指无限提供空字符的零设备，所以相当于是10000个空块

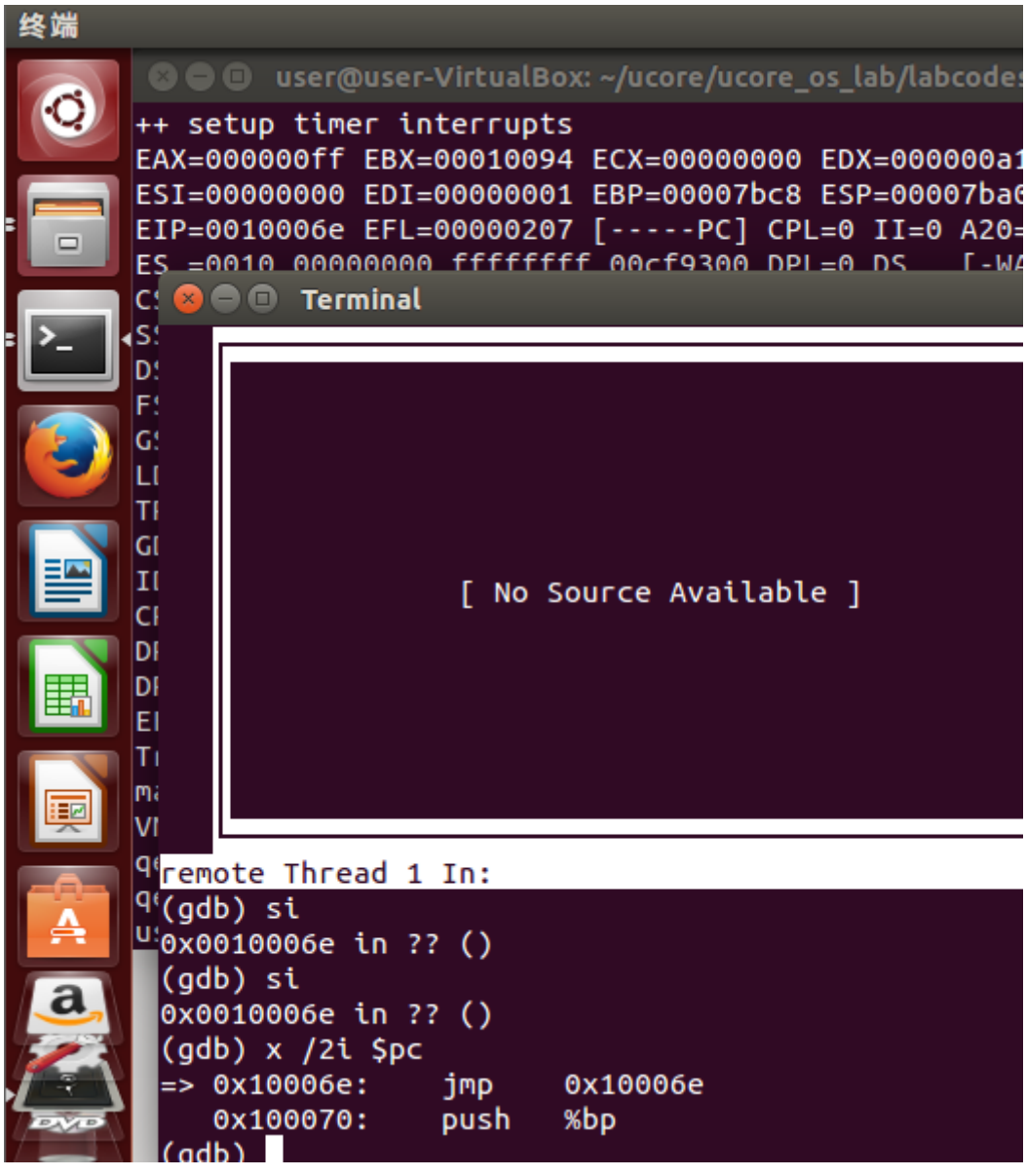
```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/stat.h>

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;
}
```

特征：主引导扇区512位，第510，511位为0x55AA

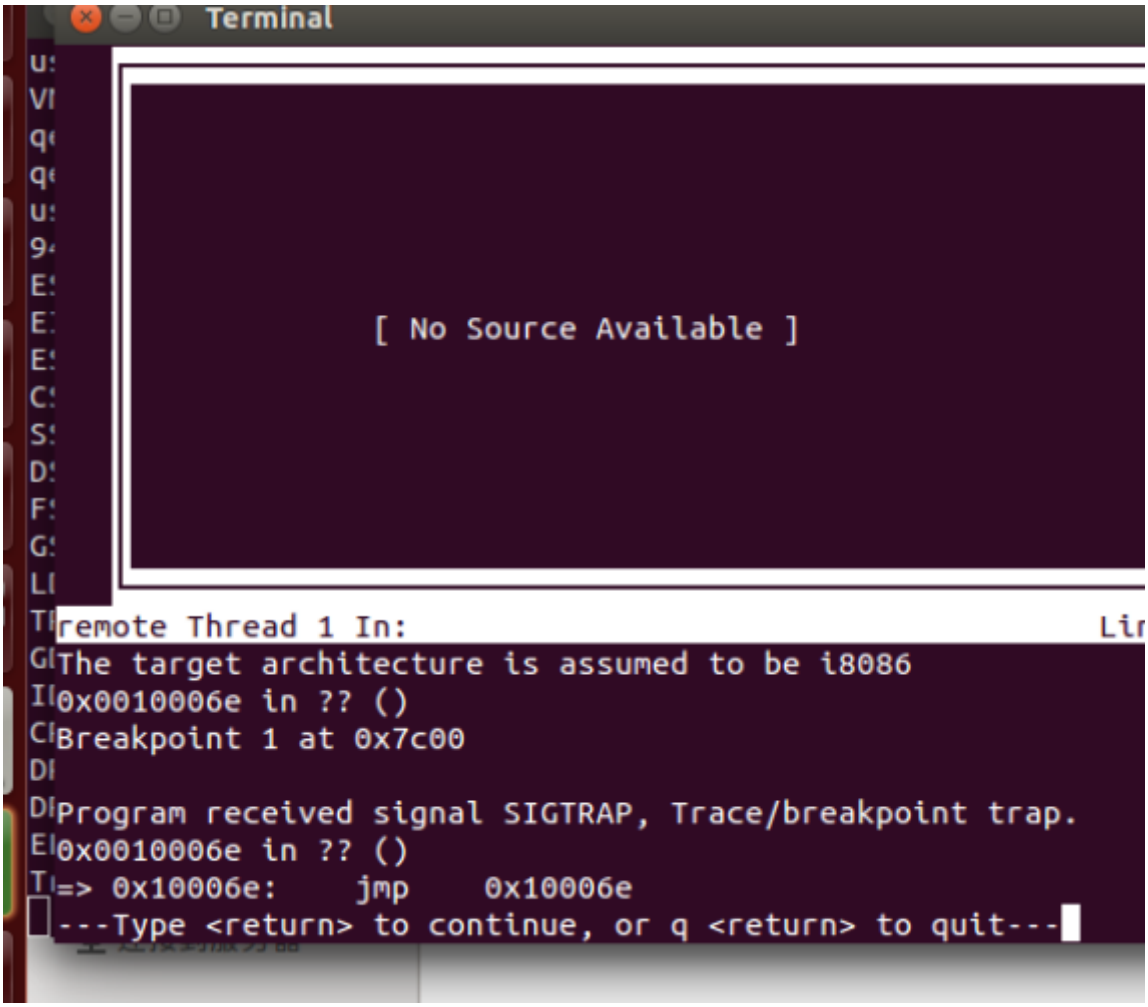
lx2

先改`gdbinit`再以`make debug`方式打开gdb



修改gdbinit，设置为以下

```
set architecture i8086//改架构
target remote :1234
b *0x7c00   //在0x7c00处设置断点。
c           //继续执行
x /2i $pc   //显示当前eip处的汇编指令
set architecture i386   //改架构
```



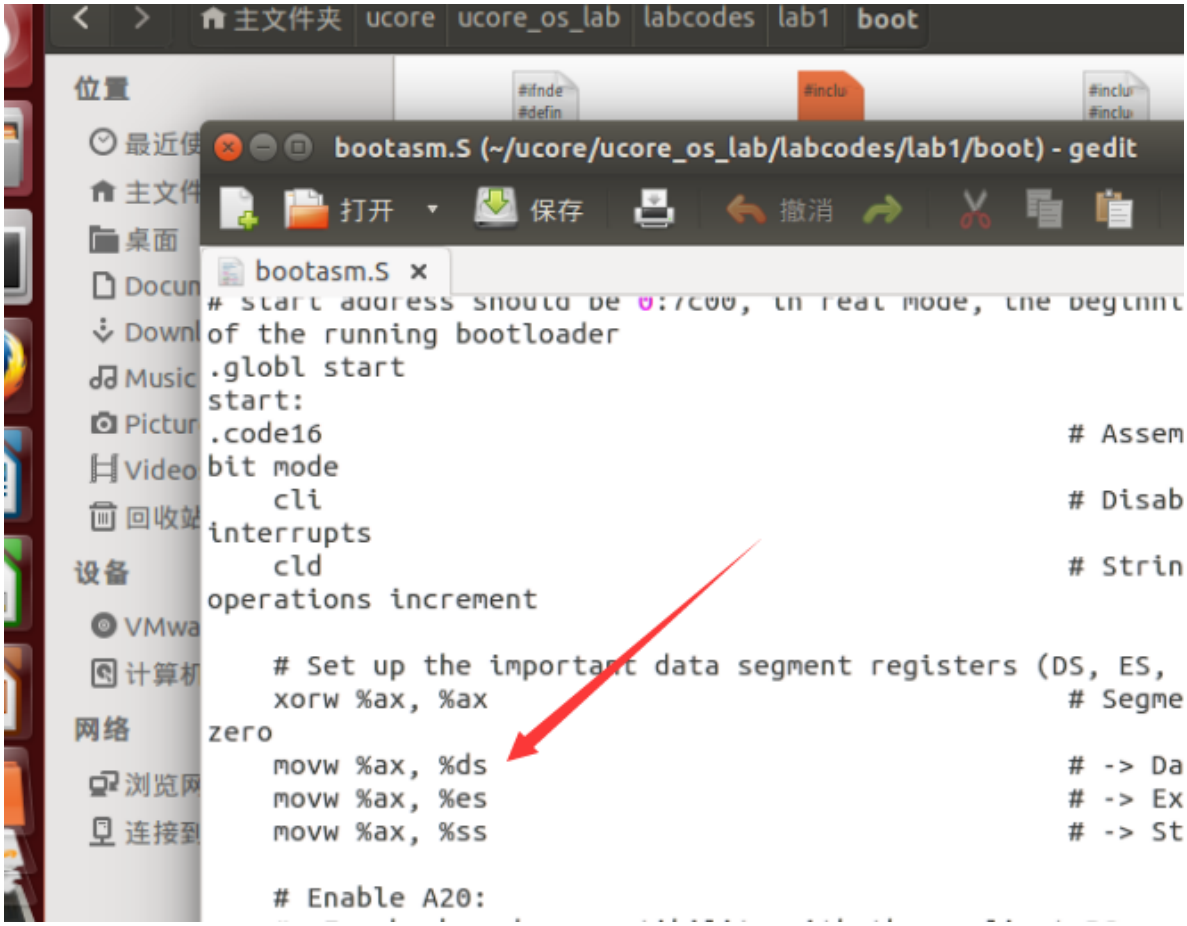
修改gdbinit

```
set architecture i8086//改架构
    target remote :1234
    b *0x7c00  //在0x7c00处设置断点。
    c          //继续执行
    x /10i $pc  //显示当前eip处的汇编指令
```

读取q.log的内容：是一系列汇编指令，其内容为从0x7c00地址开始的一系列汇编指令

lx3

逆向分析bootasm.s的代码



异或将ax置0，赋值给ds，es，ss三个寄存器，这三个分别是三个段寄存器

```
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,
```

把A20开启，A20开启模式下，可以使用32位总线，表达4G的地址空间（关闭时是20位表示1M）

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

将gdt初始化，再将cr0寄存器置1（orl：位或）

```
    # 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
```

更新基地址再重设段寄存器，最后call bootmain

lx4

elf是linux下标准的可执行文件

SECTSIZE是512字节，outb是一个用于io写入字节的函数

```
readsect(void *dst, uint32_t secno) {
    // wait for disk to be ready
    waitdisk();

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

函数的大意是从第secno扇区，逐8bit（字节）读到dst

通过readseg函数可以循环读取任意长的内容。类似于buffer机制

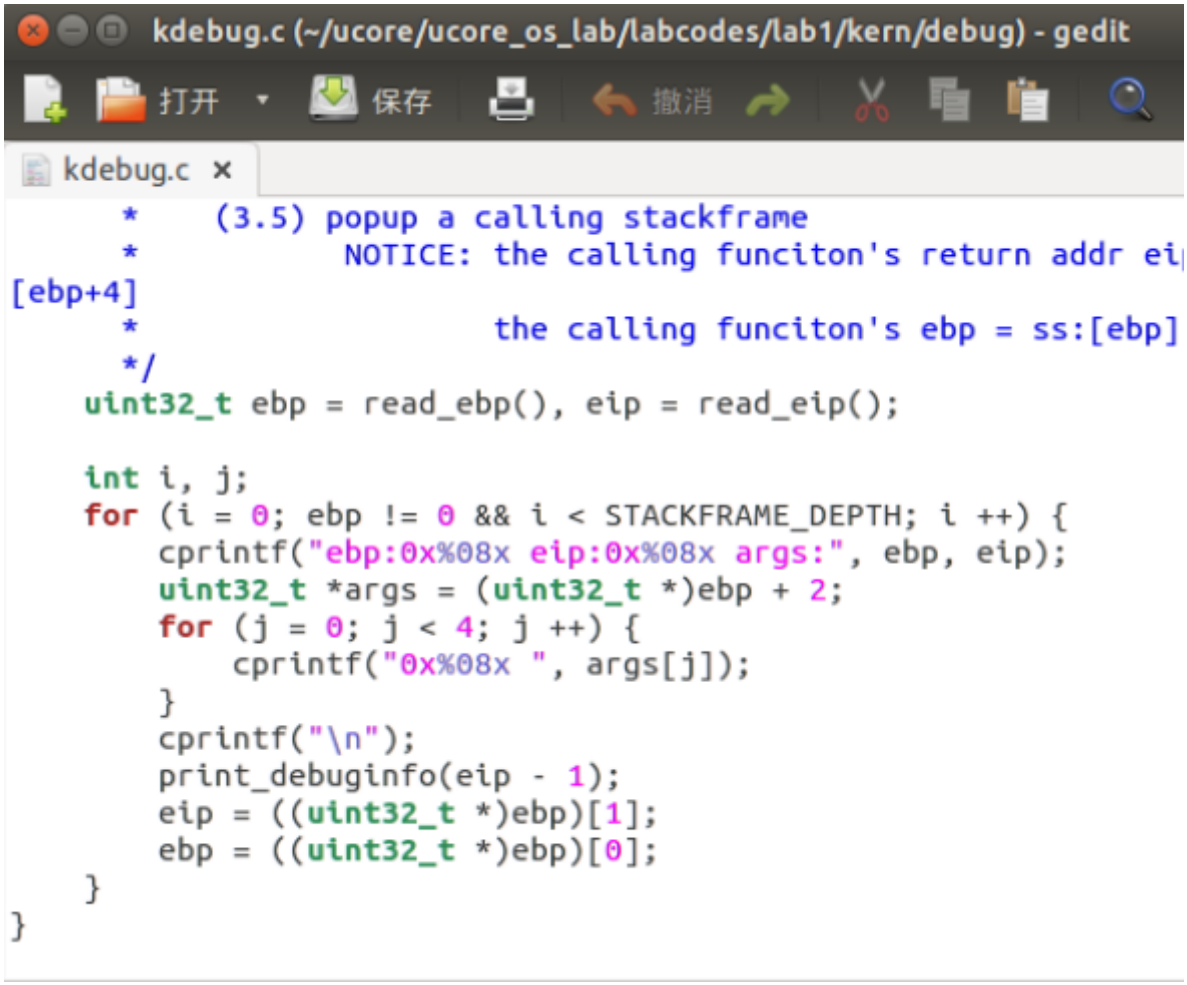
```
static void
readseg(uintptr_t va, uint32_t count, uint32_t offset) {
    uintptr_t end_va = va + count;

    // 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);
    }
}
```

分析bootmain函数，写在下面代码的注释

```
bootmain(void) {
        // 读取ELF的头部
        readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);
        // 判定文件合法性
        if (ELFHDR->e_magic != ELF_MAGIC) {
            goto bad;
        }

        struct proghdr *ph, *eph;

        // ELF头部有描述表，从描述表找到对应的加载基址
        // ph是描述表地址
        ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
        eph = ph + ELFHDR->e_phnum;

        // 载入内存
        for (; ph < eph; ph ++) {
            readseg(ph->p_va & 0xFFFFFF, ph->p_memsz, ph->p_offset);
        }
        ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();
    //不合法的情况
    bad:
        outw(0x8A00, 0x8A00);
        outw(0x8A00, 0x8E00);
        while (1);
    }
```

lx5

实现该功能需要修改kdebug.c的函数



以上内容即为实现堆栈调用跟踪的代码

该代码大意为：使用变量ebp，eip存当前读取的ebp寄存器和eip寄存器的值，在循环体中边打印边下移，将运行到的代码（对应的寄存器的值，包括args（函数的参数，考虑到栈帧的结构该位置对应函数参数））逐步打印出来

lx6

修改trap.c为以下内容

```
      */
    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);
}

static const char *
```

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

中断向量表：

中断向量表一个表项大小为8字节，其中2-3字节是段选择子，0-1字节和6-7字节拼成位移， 两者联合便是中断处理程序的入口地址。

以上代码的含义为 循环设置中断向量表（存于__vectors）的值为0（初始化），再加载IDT

trap只需要调用dispatch来触发中断即可

Challenge

在idt_init中，将用户态调用SWITCH_TOK中断的权限打开。 SETGATE(idt[T_SWITCH_TOK], 1, KERNEL_CS, __vectors[T_SWITCH_TOK], 3);

//该函数的意思是，T_SWITCH_TOK可以认为是触发中断的地址（相对偏移），将值设定为3（用户态权限）

在trap_dispatch中，将iret时会从堆栈弹出的段寄存器进行修改 以下分别是对user用户态和kernel内核态的段寄存器进行赋值

```
        tf->tf_cs = USER_CS;
        tf->tf_ds = USER_DS;
        tf->tf_es = USER_DS;
        tf->tf_ss = USER_DS;
```

```

```

```
        tf->tf_cs = KERNEL_CS;
        tf->tf_ds = KERNEL_DS;
        tf->tf_es = KERNEL_DS;
```

在lab1_switch_to_user中，调用T_SWITCH_TOU中断。 注意从中断返回时，会多pop两位，并用这两位的值更新ss,sp，损坏堆栈。 所以要先把栈压两位，并在从中断返回后修复esp。

```
    asm volatile (
        "sub $0x8, %%esp \n"
        "int %0 \n"
        "movl %%ebp, %%esp"
        :
        : "i"(T_SWITCH_TOU)
    );
```

在lab1_switch_to_kernel中，调用T_SWITCH_TOK中断。 注意从中断返回时，esp仍在TSS指示的堆栈中。所以要在从中断返回后修复esp。

```
    asm volatile (
        "int %0 \n"
        "movl %%ebp, %%esp \n"
        :
        : "i"(T_SWITCH_TOK)
    );
```

但这样不能正常输出文本。根据提示，在trap_dispatch中转User态时，将调用io所需权限降低。

```
    tf->tf_eflags |= 0x3000;
```