

OS_lab1

学号: 2011360

姓名：牟迪

代码仓库: https://github.com/KIDSSCC/NKU_OS_2022

小组成员:

2013471 宋彦艳

2013536 汤清云

2011360 牟迪

Teasing Time

Q1: 环境配置时安装qemu参照各种方法均有bug, 不确定是不是真的装上了qemu

A1: 担心多余了, 按照编译原理课程实验所配置的实验环境基本上已经满足了OS实验的需要 (暂时)

T1: 对gdb工具不熟悉, 在修改gdbinit文件时经常不知道需要预先写好哪些指令

T2: 执行make debug命令进行单步调试时，总是出错，而且身边找不到同学有一样的错误。最后发现在root用户下执行就会报错，在普通的用户下执行不会出问题，不理解。

练习1

理解make生成执行文件的过程

- 操作系统镜像文件ucore.img是如何一步一步生成的
- 一个系统被认为是符合规范的硬盘主引导扇区的特征是什么

一.ucore.img的生成

在makefile中，查找有关ucore.img的信息

[illegible]

首先调用了函数totarget，该函数来源于tools文件夹下的function.mk文件，其中有关totarget的说明：

```
totarget = $(addprefix $(BINDIR)$(SLASH),$(1))
#其中涉及两个变量，BINDIR与SLASH，经查找，二者的相关声明位于tools文件夹的Makefile文件中，
SLASH    := /
BINDIR   := bin
```

由此可以判断出，totarget函数的目的在于将生成的文件置于bin目录下。随后生成UCOREIMG所需的相关依赖即kernel和bootblock。分别阅读有关二者的信息：

```
# 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)
```

在kernel的生成过程，首先指明其保存路径，同样在bin目录下，进一步指明其所需的依赖文件，即tools目录下的kernel.ld与KOBJS。随后指令依次进行了参数列表的打印，通过链接器生成可执行文件，随后通过objdump工具来反汇编解析得到符号表。最终目的链接生成文件bin/kernel

```
# 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)

# -----

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

# -----
```

通过指令 make “V=”来获取编译过程中的更为具体的信息，关于bootblock的生成有如下操作。

```

+ 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: 492 bytes
build 512 bytes boot sector: 'bin/bootblock' success!

```

相关编译参数	
-Iboot/	指定头文件的查找路径为boot文件夹
-march	指定编译的CPU架构，此处为i686
-fno-builtin	不进行以_builtin_函数的优化
-fno-PIC	不产生绝对地址，使代码可以被随意加载
-Wall	显示所有警告信息
-ggdb	生成用于gdb的调试信息
-m32	生成32位机器代码
-gstabs	以stabs格式生成调试信息
-nostdinc	只在-I选项确定的文件夹中寻找头文件
-fno-stack-protector	不生成用于检测缓冲区溢出的代码

通过具体的编译过程可以发现，bootblock的生成过程需要bootasm.S，bootmain.c，sign.c文件，将bootasm.S汇编为bootasm.o，将bootmain.c编译为bootmain.o，将sign.c编译为bin目录下的sign文件。最终将bootmain.o和bootasm.o链接得到bootblock.o文件，最终得到bootblock文件

在完成相关的依赖后，ucore.img的生成过程，首先通过\dev\zero读取一定数目的空字符。随后将编译好的bootblock和kernel写入到ucore.img中。因此得到总的生成过程

1. 编译链接kernel所需要的文件
2. 生成bin\kernel
3. 编译bootasm.S，bootmain.c，sign.c
4. 生成bin\bootblock
5. 生成ucore.img

二.规范的硬盘主引导扇区

主引导扇区即bootblock被加载到的区域，在程序sign.c中查找有关信息

```
char buf[512];
memset(buf, 0, sizeof(buf));
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;
}
```

由此可判断主引导扇区的要求：大小为512字节，第510个字节为0x55，第511个字节为0xAA

练习2

- 从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行
- 从初始化位置0x7c00设置实地址断点，测试断点正常
- 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。
- 自己找一个bootloader或内核中的代码位置，设置断点并进行测试。

一.单步跟踪第一条指令

修改gdbinit文件

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

执行make debug命令

```
QEMU [Paused]
Machine View
Terminal
> 0xffff0 add %al, (%eax)
0xffff2 add %al, (%eax)
0xffff4 add %al, (%eax)
0xffff6 add %al, (%eax)
0xffff8 add %al, (%eax)
0xffffa add %al, (%eax)
0xffffc add %al, (%eax)
0xffffe add %al, (%eax)
0x10000 add %al, (%eax)
0x10002 add %al, (%eax)
0x10004 add %al, (%eax)
0x10006 add %al, (%eax)
0x10008 add %al, (%eax)

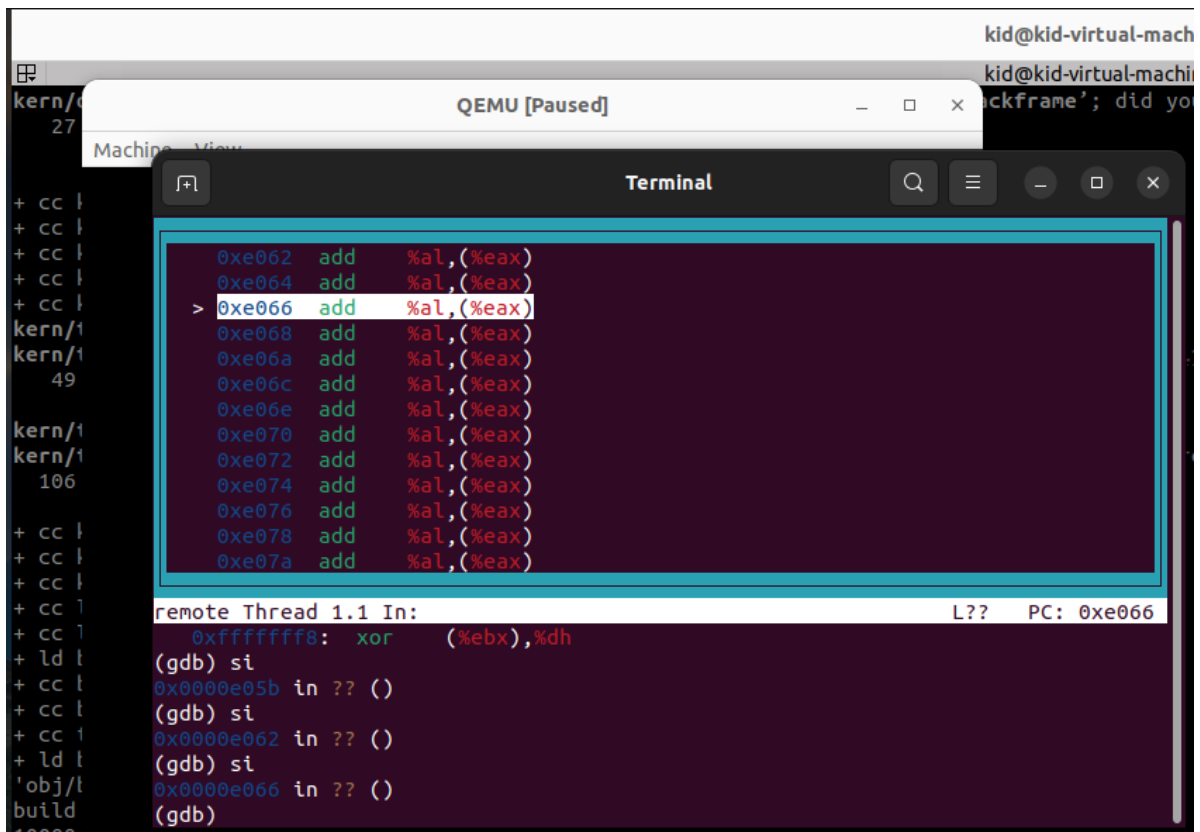
remote Thread 1.1 In: L?? PC: 0xffff0
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x0000ffff in ?? ()
The target architecture is set to "i8086".
(gdb) p /x $eip
$1 = 0xffff0
(gdb)
```

此时CPU加电，在第一条指令前停下，打印eip寄存器的值，可发现eip的值为0xffff0，在开机复位时，CS段寄存器将被设置为0xffff，因此，在Segment:Offset表示下，真正执行的第一条指令地址为：0xFFFFFFF0，通过gdb调试可以得到对应地址的指令

```
QEMU [Paused]
Machine View
Terminal
> 0xffff0 add %al, (%eax)
0xffff2 add %al, (%eax)
0xffff4 add %al, (%eax)
0xffff6 add %al, (%eax)
0xffff8 add %al, (%eax)
0xffffa add %al, (%eax)
0xffffc add %al, (%eax)
0xffffe add %al, (%eax)
0x10000 add %al, (%eax)
0x10002 add %al, (%eax)
0x10004 add %al, (%eax)
0x10006 add %al, (%eax)
0x10008 add %al, (%eax)

remote Thread 1.1 In: L?? PC: 0xffff0
The target architecture is set to "i8086".
(gdb) p /x $eip
$1 = 0xffff0
(gdb) x/3i 0xfffffff0
0xfffffff0: ljmp $0x3630, $0xf000e05b
0xfffffff7: das
0xfffffff8: xor (%ebx), %dh
(gdb)
```

因此程序执行的第一条指令为ljmp长跳转指令，通过此条指令，一方面将CS寄存器的值更新为0xf000，另一方面将eip设置为0xe05b，跳转至BIOS例行程序起点进行初始化，完成初始化后，会跳转到地址0x7c00，即写入bootloader的地址，并将CPU控制权转交给bootloader。



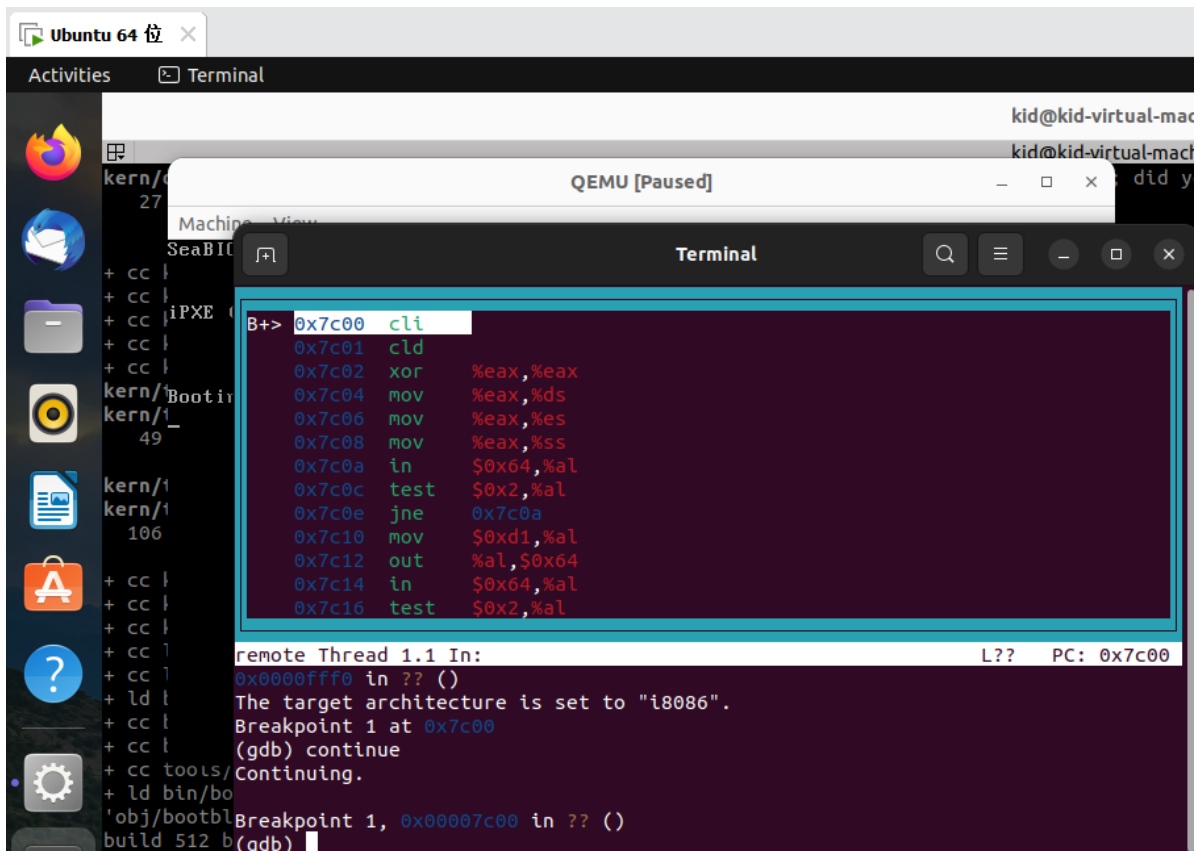
通过si指令可以实现单步执行，可发现指令跳转到0x0000e05b开始执行

二.bootloader执行

修改gdbinit文件，添加断点：

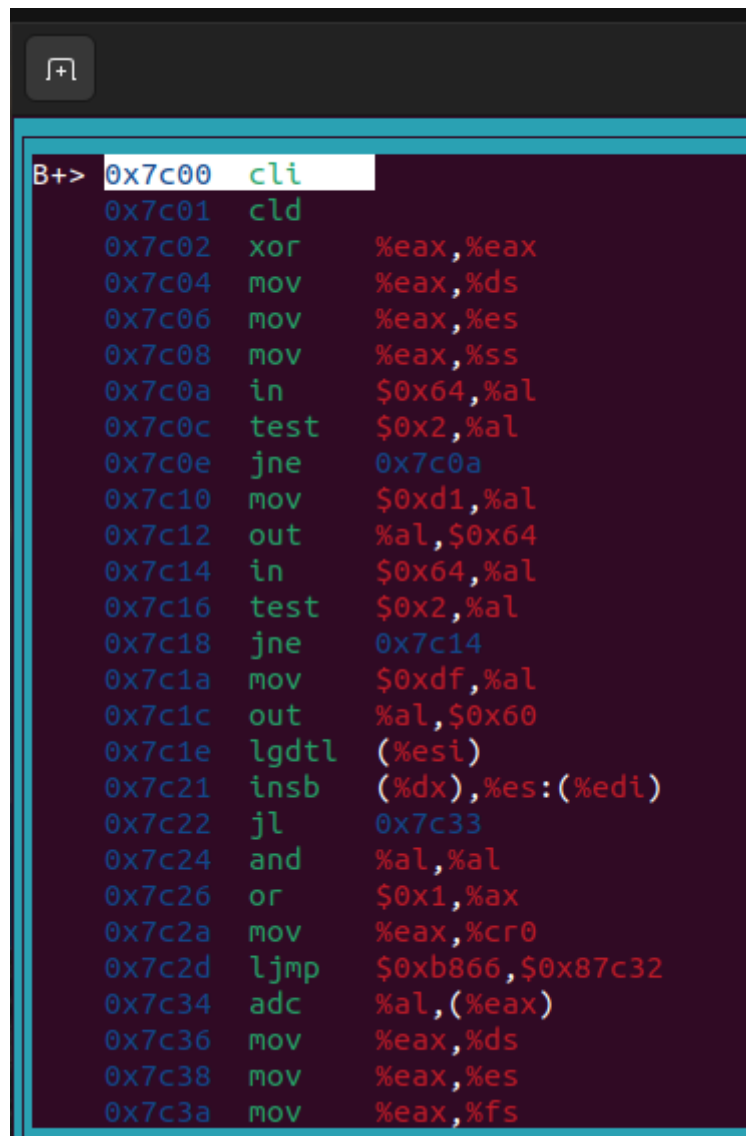
```
b *0x7c00
```

在进行gdb调试的过程中通过continue命令即可运行到0x7c00地址处



三.代码分析

通过二的操作，此时可观察由地址0x7c00开始的指令



```
B+> 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
0x7c1c out %al,$0x60
0x7c1e lgdtl (%esi)
0x7c21 insb (%dx),%es:(%edi)
0x7c22 jl 0x7c33
0x7c24 and %al,%al
0x7c26 or $0x1,%ax
0x7c2a mov %eax,%cr0
0x7c2d ljmp $0xb866,$0x87c32
0x7c34 adc %al,(%eax)
0x7c36 mov %eax,%ds
0x7c38 mov %eax,%es
0x7c3a mov %eax,%fs
```

在编译阶段，会将bootasm.S进行编译得到obj\bootblock.o文件，同时，在Makefile中对其进行了反汇编得到obj\bootblock.asm文件。在主要的代码上，二者是完全一样的，区别在于经过编译链接后，在bootblock.asm文件中将增加了更为具体的指令地址信息，即从0x7c00地址开始，而在调试过程中，0x7c00地址处开始的指令与bootasm.S和bootblock.o相同。

四.断点测试

具体测试见练习5

练习3

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位置执行bootloader，请分析bootloader是如何

完成从实模式到保护模式的转换

- 为何开启A20，以及如何开启A20
- 如何初始化GDT表
- 如何使能和进入保护模式

计算机加电后，不会直接运行操作系统，而是先对系统软件进行初始化，完成基本IO初始化和引导加载对于Intel 80386体系结构，初始化工作由BIOS和bootloader一起组成，BIOS做完系统自检和初始化后，会将硬盘主引导扇区到读取到内存，跳转到对应地址并将CPU控制权转移到对应的地址，开始执行bootloader

一.A20

为何要开启A20:

本质是向下兼容所产生的遗留问题。在8086中，地址线20位，用于寻址1M的内存空间，而寄存器的大小为16位，不能满足寻址需求，因此在8086中，采用一个寄存器表示基址，进行移位后与另一表示偏移量的寄存器相加。在8086中，内存大小增至4G，寄存器大小也变为了32位，为了使以前的机器也能使用这种方式，就以第21根地址线作为开关，当其被使能时，是一根正常的地址线，此时机器可视为处于保护模式，可寻址全部的4G内存。而当其为0时，机器处于实模式，可寻址内存仅为1M，因此，为实现从实模式到保护模式的转换，需要开启A20.

如何开启A20:

A20由8042键盘控制器来控制，8042芯片中的Output Port端口中第一位即表示A20

两个端口，0x64与0x60.

```
seta20.1:
    #确定当前输入缓存中没有数据才可以进行下一步操作
    inb $0x64, %a1                                # 从64h端口读取当前状态，存储到a1寄存器
    testb $0x2, %a1                                # 进行与运算，根据与运算结果设置状态位，这里用于判断第二位是否为1
    jnz seta20.1                                    # 第二位为1，说明缓存不为空，返回重试
    #准备写入，向0x64端口传递准备写Output Port的信号
    movb $0xd1, %a1                                #向0x64写入0xd1，表示当前要对Output Port进行写操作
    outb %a1, $0x64
```

经如上小节的代码，bootloader准备向控制A20的8042芯片进行写操作

```
seta20.2:
    #与上一小节相同，确保缓存中无数据
    inb $0x64, %a1
    testb $0x2, %a1
    jnz seta20.2
    #向0x60端口传递要写入的数据
    movb $0xdf, %a1
    outb %a1, $0x60
```

经过如上两小节代码的执行，A20被开启，程序可寻址4G内存

二.如何初始化GDT表

GDT即全局描述符表，保存在专门的段寄存器gdtr中，GDT的初始化通过如下一句代码实现:

```
lgdt gdtdesc
```


lgdt即初始化段寄存器gdt，其操作数为48位，对应48位的gdt寄存器，其中32位用于描述段表起始地址，16位用以描述其范围。通过阅读gdt desc的定义：

```
gdt desc:
    .word 0x17                                # sizeof(gdt) - 1
    .long gdt                                # address gdt
```

0x17即表示其大小范围，起始地址由gdt给出，而在gdt中包含三部分：

```
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
```

分别用以描述空段，代码段和数据段。相关段的具体空间分配在文件asm.h给出

三.如何使能和进入保护模式

对于保护模式的转换设计cr0寄存器，cr0中含有控制处理器模式和状态的系统标志位，其位0即启用保护标志，当其有效时即进入保护模式。通过如下指令：（其中\$CR0_PE_ON为预先声明，其值为1）将cr0寄存器中位0置1，从而进入保护模式。

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

练习4

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来运行和调试bootloader&OS

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

一.读取硬盘扇区

通过bootasm.c部分代码的执行，CPU已经进入保护模式，下一步的工作即将OS从硬盘中加载到内存中并运行。bootloader的硬盘访问采用LBA模式的Program IO方式，所有的IO操作都通过CPU访问硬盘的IO地址寄存器来完成。例如访问第一个硬盘的扇区通过设置IO地址寄存器0x1F0-0x1F7来实现

```
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);
    }
}

```

通过readseg函数，从offset地址处读取了count个字节并保存在虚拟地址va处，并把值传递给readsect函数

```

static void
waitdisk(void) {
    while ((inb(0x1F7) & 0xC0) != 0x40)
        /* do nothing */;
}

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

    outb(0x1F2, 1);                // count = 1 读取一个扇区
    outb(0x1F3, secno & 0xFF);      //0-7位
    outb(0x1F4, (secno >> 8) & 0xFF); //8-15位
    outb(0x1F5, (secno >> 16) & 0xFF); //16-23位
    outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0); //24-27位
    outb(0x1F7, 0x20);              // cmd 0x20 - read sectors

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

    // read a sector
    insl(0x1F0, dst, SECTSIZE / 4);
}

```

读一个扇区的工作主要由如上两个函数来完成。在函数waitdisk中，寄存器0x1F7为状态和命令寄存器，代表当前磁盘的工作状态，当磁盘处于忙状态时，则进行等待。

在readsect函数中。首先等待磁盘完成准备工作进入空闲状态，然后依次向IO地址寄存器进行写入，用以描述当前要进行的读取动作。在LBA模式下，寄存器0x1F1默认为0，不进行任何操作，0x1F2代表要读取的扇区数量，此处要读取一个扇区。0x1F3-0x1F6开始按位传递读取磁盘所需的LBA参数。其中0x1F6寄存器的第5位还代表了当前的磁盘访问模式（当前为LBA）向0x1F7寄存器所传递的参数0x20则代表了当前要进行的是一次读操作。最终，通过函数insl完成数据的读取，读取到的数据通过0x1F0寄存器返回，每次一个字

二.加载ELF格式的OS

在主函数中，完成磁盘的读取后，便进行OS的加载

```

    //检查读取到的内容的e_magic位是不是ELF_MAGIC,如果不是，说明当前文件存在问题，转至bad处。
    否则进行ELF文件的加载
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }
}

```

```

//ph为数据表的起始位置，eph是数据表的结束位置
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 & 0xFFFFFFFF, ph->p_memsz, ph->p_offset);
}

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

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

    /* do nothing */
    while (1);

```

通过代码中可以得到，ph代表了数据表的起始位置，初始通过基址+表的位置偏移确定（e_phoff）eph则代表了结束地址，e_phnum代表了表中的入口数目，在for循环中，循环把数据装入内存。p_va代表了第一个字节将被存入的虚拟地址，p_memsz代表了在内存中所占用的字节数，p_offset是其相对于文件头的偏移地址。

因此，可得bootloader加载ELF文件的步骤：

1. 判断文件是否有效
2. 确定数据表起始位置与终止位置
3. 根据每个段要存入的起始地址和段的大小，将各个段存入内存

练习5

在kdebug中完成函数print_stackframe的实现，通过函数print_stackframe来跟踪函数调用堆栈中记录的返回地址。

在kern/debug/kdebug.c文件中，根据注释信息补全函数print_stackframe

```

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];
}

```

补充函数后，通过make qemu指令进行调试，得到如下结果：

```

Special kernel symbols:
  entry 0x00100000 (phys)
  etext 0x0010350b (phys)
  edata 0x0010fa16 (phys)
  end   0x00110d08 (phys)
Kernel executable memory footprint: 68KB
ebp=: 0x00007b28 | eip=: 0x001009a5 | args=: 0x00010094 0x00010094 0x00007b58 0x0010008e
  kern/debug/kdebug.c:305: print_stackframe+21
ebp=: 0x00007b38 | eip=: 0x00100c9c | args=: 0x00000000 0x00000000 0x00000000 0x00007ba8
  kern/debug/kmonitor.c:125: mon_backtrace+10
ebp=: 0x00007b58 | eip=: 0x0010008e | args=: 0x00000000 0x00007b80 0xfffff000 0x00007b84
  kern/init/init.c:48: grade_backtrace2+33
ebp=: 0x00007b78 | eip=: 0x001000bc | args=: 0x00000000 0xfffff000 0x00007ba4 0x00000029
  kern/init/init.c:53: grade_backtrace1+40
ebp=: 0x00007b98 | eip=: 0x001000dc | args=: 0x00000000 0x00100000 0xfffff000 0x0000001d
  kern/init/init.c:58: grade_backtrace0+23
ebp=: 0x00007bb8 | eip=: 0x00100104 | args=: 0x0010353c 0x00103520 0x000012f2 0x00000000
  kern/init/init.c:63: grade_backtrace+34
ebp=: 0x00007be8 | eip=: 0x00100051 | args=: 0x00000000 0x00000000 0x00000000 0x00007c4f
  kern/init/init.c:28: kern_init+80
ebp=: 0x00007bf8 | eip=: 0x00007d72 | args=: 0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
<unknow>: -- 0x00007d71 --

```

自底向上代表了函数的调用顺序与栈帧的切换顺序，通过gdb单步调试来进行验证，修改gdbinit文件：

```

file bin/kernel
set architecture i8086
target remote :1234
b kern_init
continue

```

执行make debug命令进入单步调试阶段。可发现函数最开始的调用是在kern_init函数，在此处插入断点。

```
Terminal
kern/init/init.c
23     const char *message = "(THU.CST) os is loading ...";
24     cprintf("%s\n\n", message);
25
26     print_kerninfo();
27
> 28     grade_backtrace();
29
30     pmm_init();           // init physical memory management
31
32     pic_init();           // init interrupt controller
33     idt_init();           // init interrupt descriptor table
34
35     clock_init();         // init clock interrupt

remote Thread 1.1 In: kern_init      L28    PC: 0x10004c
(gdb) n
(gdb) n
(gdb) p /x $ebp
$1 = 0x7be8
(gdb) x /8x $ebp
0x7be8: 0x00007bf8      0x00007d72      0x00000000      0x00000000
0x7bf8: 0x00000000      0x00007c4f      0xc031fcfa      0xc08ed88e
(gdb)
```

在进入grade_backtrace前，查看ebp的值与相应地址的值。可发现ebp为0x7be8，在内存中其+4后地址所存数据即为上一栈帧的ebp值（push ebp的结果）。+4后得到返回地址即0x7dd2即调用时的eip，后续四个四字节的数据分别对应函数调用时传入的参数即args所对应的值。与打印结果相对应，进入函数继续执行，结果符合。

对于最后一行参数，ebp即调用函数kern_init时原栈帧ebp的地址，eip为指令指针寄存器，存储了下一条要执行的指令，同时也代表了跳转时的返回地址。args为对应空间中的输入参数，在此处为bootloader的代码段。

练习6

- 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？
- 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。
- 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”。

一.中断描述符表

一个中断表项占8个字节，其中第2, 3字节代表段选择子，第0, 1字节与4, 5, 6, 7四个字节拼接得到偏移量，二者共同得到中断处理代码的入口。在mmu.h文件中，可查找到有关中断描述符的详细定义：

```

/* Gate descriptors for interrupts and traps */
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
};

```

二.补充trap.c

在kern/trap.c中可发现，中断描述符表idt为大小256的类型为gatedesc的数组，256个中断处理的地址已由tools/vector.c生成，其详细结果保存在kern/vector.s中，在trap.c中，可直接通过__vectors[]进行调用。补充代码如下：

```

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
     *      __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 __vectors[];
    for(int i=0;i<sizeof(idt)/sizeof(struct gatedesc);i++){
        SETGATE(idt[i],0,GD_KTEXT,__vectors[i],DPL_KERNEL);
    }
    SETGATE(idt[T_SWITCH_TOK],0,GD_KTEXT,__vectors[T_SWITCH_TOK],DPL_KERNEL);
    lidt(&idt_pd);
}

```

有关SETGATE函数的定义位于mm/mmu.h文件中，具体声明如下：

```

#define SETGATE(gate, istrap, sel, off, dpl) {           \
    (gate).gd_off_15_0 = (uint32_t)(off) & 0xffff;    \
    (gate).gd_ss = (sel);                               \
    (gate).gd_args = 0;                                 \
    (gate).gd_rsv1 = 0;                                 \
    (gate).gd_type = (istrap) ? STS_TG32 : STS_IG32;    \
    (gate).gd_s = 0;                                    \
    (gate).gd_dpl = (dpl);                             \
    (gate).gd_p = 1;                                    \
    (gate).gd_off_31_16 = (uint32_t)(off) >> 16;      \
}

```

通过该语句块可将中断描述符进行初始化，其中的各个参数：

参数	含义
gate	要进行初始化的描述符变量gatedesc
istrap	0或者1，0代表中断门，1代表陷阱门，在此处选择中断门
sel	中断处理程序所在的段选择子，此处为GD_KTEXT，具体定义位于mm/memlayout.h，代表kernel的代码段
off	32位偏移量，由数组_vector[]提供
dpl	对应中断处理程序的特权级，此处为DPL_KERNEL，具体定义在mm/memlayout.h处，代表0

值得注意的是，中断描述符表中T_SWITCH_TOK（声明位于文件trap.h）是由user发起的中断，不是由kernel所发起，因此对其单独初始化。完成初始化工作后，最后一步工作即将初始化好的idt表加载到idtr寄存器中，通过调用函数lidt实现，lidt函数具体定义位于libs/x86.h

三.补充时钟中断处理

kern/trap/trap.c中的trap_dispatch函数内声明了对于不同异常的处理。在trap.h文件中，可以得到相关的编号声明，IRQ_OFFSET + IRQ_TIMER即时间中断，TICK_NUM已被预先定义为100，可通过对ticks的统计，在发生100次中断时，调用print_ticks函数进行信息的打印。

```

ticks ++;
if (ticks % TICK_NUM == 0) {
    print_ticks();
}

```