

操作系统实验 1

17343050 黄昱琿 软件工程

练习 1 - 理解通过 make 生成执行文件的过程

```
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
```

这条命令将内核的所有对象文件全部链接了起来，而且规定不链接 C 库，架构为 32 位 x86。这个可执行文件（是 ELF 格式的）将由我们的主引导程序执行。

```
ld -m elf_i386 -nostdlib -N -e start -Ttext 0x7c00 obj/boot/bootasm.o
obj/boot/bootmain.o -o obj/bootblock.
```

这条命令链接得到主引导记录，设置了代码段 start 的偏移地址为 0x7c00 以保证满足约束条件。生成的可执行文件将由 QEMU 负责调用。

```
build 512 bytes boot sector: 'bin/bootblock' success!
dd if=/dev/zero of=bin/ucore.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.177799 s, 28.8 MB/s
dd if=bin/bootblock of=bin/ucore.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.0030148 s, 170 kB/s
dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
146+1 records in
146+1 records out
74828 bytes (75 kB, 73 KiB) copied, 0.0095638 s, 7.8 MB/s
```

dd 命令首先将内核映像清零，规定大小为 10000 个扇区，将便已生成的可执行文件（包括主引导记录和操作系统）生成映像供 QEMU 使用。

1. 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？

引导系统必须具备如下因素：加载到 0x7c00 处、大小为 512 字节、最后两个字节为 55 和 aa。查看 `sign.c`：

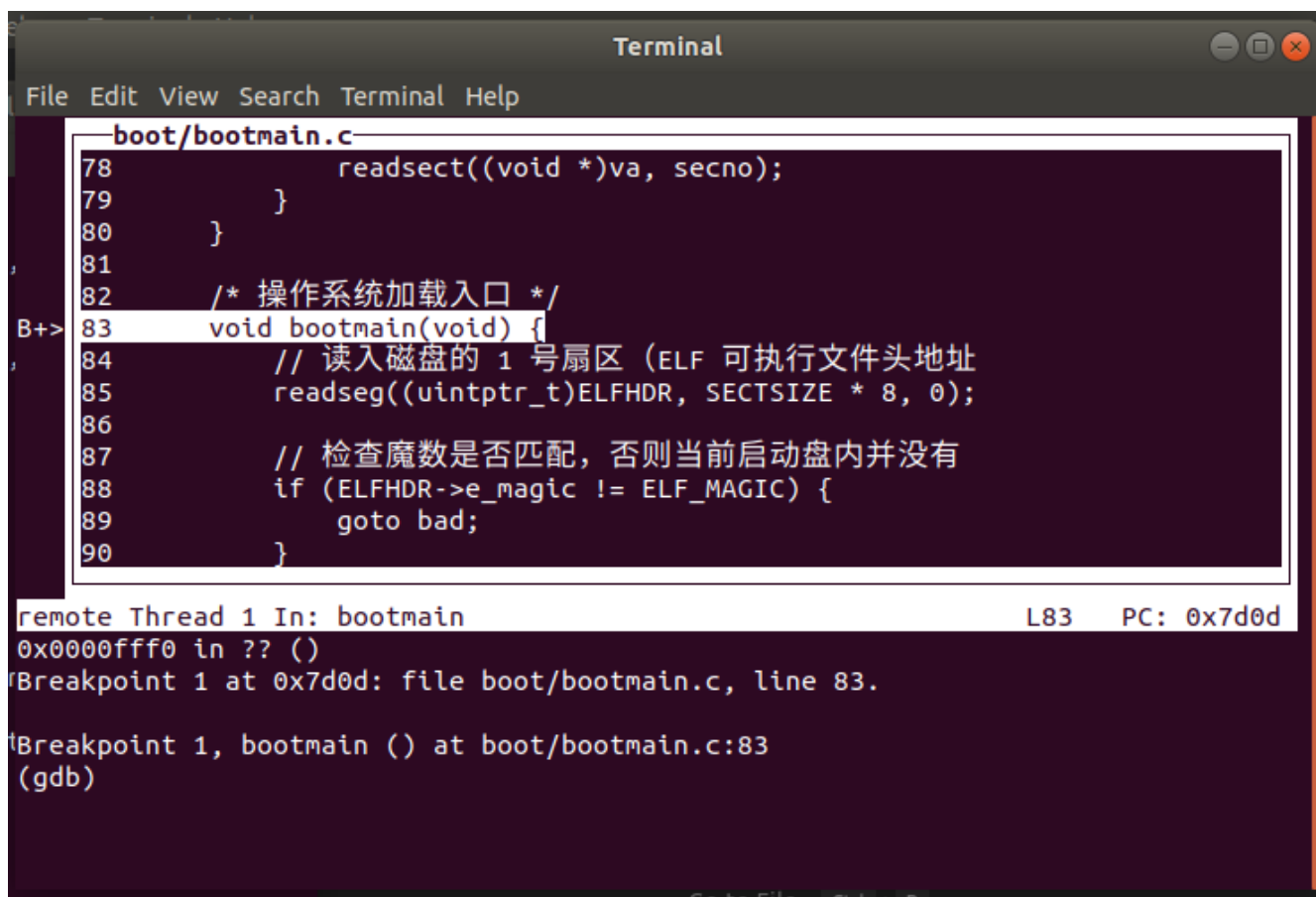
```
char buf[512]; buf[510] = 0x55; buf[511] = 0xAA;
```

buf 用来存放主引导记录的代码，设置 0x55、0xAA 后便可知其是主引导扇区。

练习 2 - 使用 QEMU 执行并调试 lab1 中的软件

1. 通过执行 `make debug` 即可开始调试，修改 `gdbinit` 如下：

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



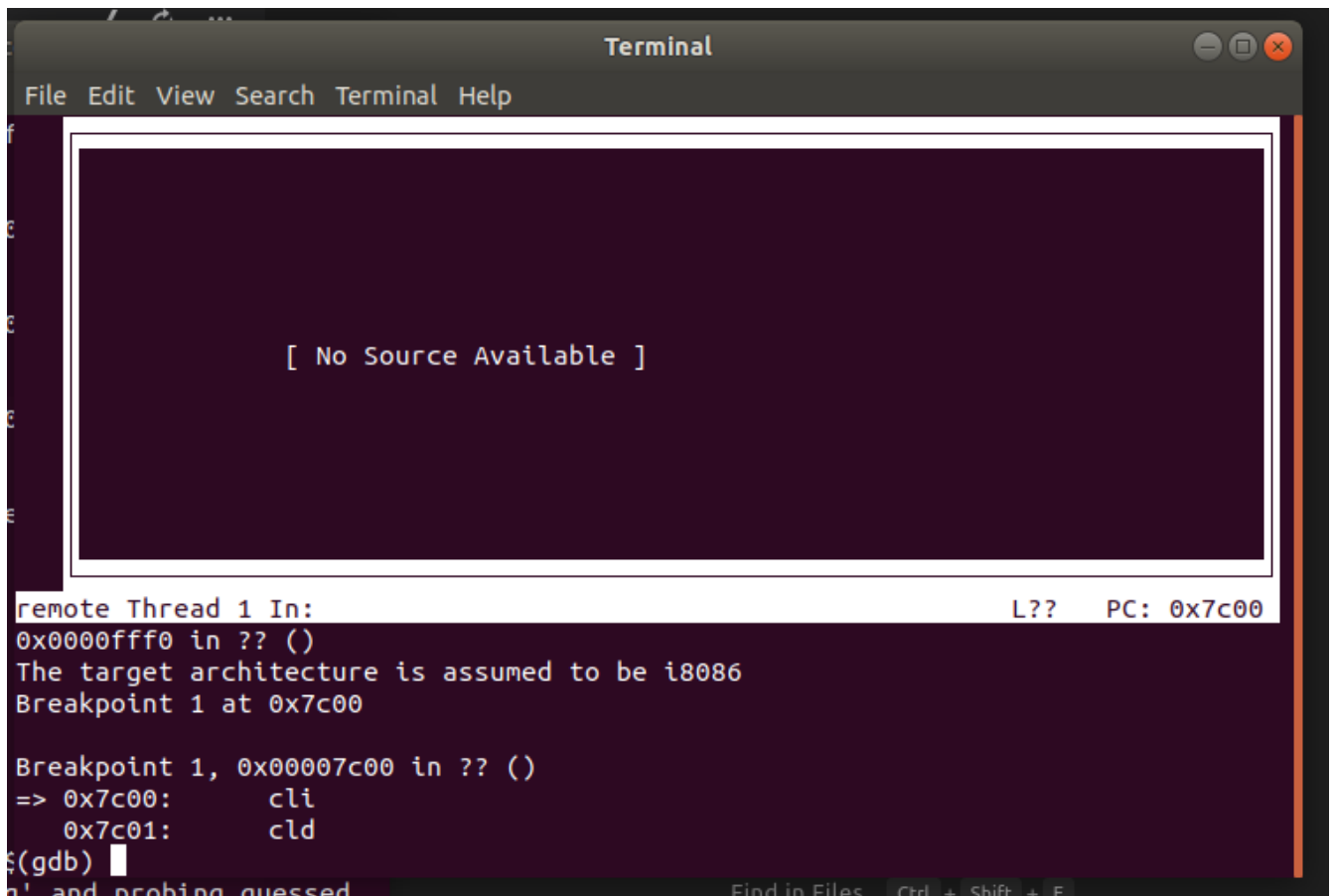
The screenshot shows a GDB terminal window with the following content:

```
Terminal
File Edit View Search Terminal Help
boot/bootmain.c
78     readsect((void *)va, secno);
79     }
80     }
81
82     /* 操作系统加载入口 */
B+> 83     void bootmain(void) {
84         // 读入磁盘的 1 号扇区 (ELF 可执行文件头地址
85         readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);
86
87         // 检查魔数是否匹配，否则当前启动盘内并没有
88         if (ELFHDR->e_magic != ELF_MAGIC) {
89             goto bad;
90         }
remote Thread 1 In: bootmain                                L83    PC: 0x7d0d
0x0000fff0 in ?? ()
Breakpoint 1 at 0x7d0d: file boot/bootmain.c, line 83.
Breakpoint 1, bootmain () at boot/bootmain.c:83
(gdb)
```

然后就可以单步调试了。

2. 修改 `gdbinit` 如下：

```
file bin/kernel
target remote :1234
set architecture i8086
b *0x7c00
continue
x /2i $pc
```



3. 经过调试知道两者相符。

练习 3 - 分析 bootloader 进入保护模式的过程

`bootasm.s` 汇编程序是 BIOS 上电后最初执行的在磁盘上的代码，存放在硬盘的主引导扇区内，所有的代码数据将被加载到内存的 `0:7c00` 中。

1. 首先禁用中断：

```
.global start
start:
.code16
cli
cld
```

2. 对 `%ax`（累加器寄存器 Accumulator）、`%ds`（数据段寄存器 Data Segment Register）、`%es`（附加段寄存器 Extra Segment Register）、`%ss`（堆栈段寄存器 Stack Segment Register）清零。
3. 通过打开 A20 GATE 来启用 32 位寻址模式。Intel 在 8086 CPU 时代提供了 20 根地址线供访问 1MB 的内存空间，但是 16 位 CPU 无法直接计算 20 位的地址。Intel 为解决该问题采取了将内存分段的技术。但是到了 80386 中，CPU 可以访问 4GB 的内存，为了向下兼容而默认处于 8086 实模式，因此为了允许 80386 寻址 32 位内存，就需要借助一个开关。IBM 利用键盘控制器多余的一根线来设置第 20 位地址线是否启用。

```
# A20地址线由键盘控制器 8042 进行控制
# 将P21引脚置1的操作：查手册知，首先要先向64h发送0xd1的指令，然后向60h发送0xdf的指令
```

```

seta20.1: # 向 64H 发送 0xD1 指令
    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: # 向 60H 发送 0xDF 指令
    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

```

4. 从实模式切换到保护模式，初始化 GDT 表

```

lgdt gdt_desc # 设置 GDTR 指向我们的 GDT

# 下面三条语句将 cr0 标志寄存器设置上 PE_ON，也就是打开保护模式的标记
movl %cr0, %eax
orl $CR0_PE_ON, %eax # 由于 orl 等运算指令不能直接操作，需要借助寄存器完成运算
movl %eax, %cr0

.p2align 2
gdt:                                     # 全局描述符表（段描述符表）
                                         # GDT 内部按顺序排放段描述符

    SEG_NULLASM                          # 空段
    SEG_ASM(STA_X | STA_R, 0x0, 0xFFFFFFFF) # 代码段，操作系统全可读，基址为 0
    SEG_ASM(STA_W, 0x0, 0xFFFFFFFF)        # 数据段，操作系统全可写，基址为 0

gdt_desc:                               # 段寄存器描述符，包含表大小及表地址，供 lgdt 指令
    .word 0x17 # sizeof(gdt) - 1
    .long gdt

```

SEG_ASM 等宏在 `asm.h` 中有所定义：

```

#define STA_X 0x08 // 可以执行代码的段
#define STA_R 0x02 // 对于可以执行代码的段，则表示段可读
#define STA_W 0x02 // 对于数据段，则表示段可写，数据段必可读
#define SEG_NULLASM .word 0, 0; .byte 0, 0, 0, 0
#define SEG_ASM(type, base, limit)

```

SEG_ASM 宏描述了一个 GDT 描述符：

63:56		55:52		51:48		47:40		39:32		31:16		15:0
Base[24:31]		Flags		Limit[16:19]		Access Byte		Base[16:23]		Base[0:15]		Limit[0:15]

5. 切换到 32 位模式。因为我们在标记了 `protcseg` 段之后全是 32 位的指令，因此跳转到 32 位指令段会直接导致 CPU 自动进入 32 位模式。由于我们已经在保护模式下，`PROT_MODE_CSEG` 表示的是段选择符。段选择符的数据结构是（`INDEX[15:3]`、`TI[2]`、`RPL[1:0]`）。由于我们现在是在启动操作系统的阶段，因此特权级 `RPL` 设为最高（即 `0b00`）。`INDEX` 为段描述表的索引，我们希望选择 GDT 的代码段（我们要进行跳转，指令由代码段寄存器寻址），因此将 `INDEX` 设为 `0x01`。`TI` 表示选择 GDT 还是 LDT，我们这里需要 CPU 在 GDT 内寻找段描述表，因此将 `TI` 设为 0（0 对应 GDT、1 对应 LDT）。因此我们现在要跳转到 32 位的代码段，因此我们要跳转到的地址是 `0008:$protcseg`。

```
ljmp $PORT_MODE_CSEG, $protcseg

.code32
protcseg:
```

6. 将各个段寄存器：`%ds`（数据段寄存器）、`%es`（附加段寄存器）、`%fs`（附加段寄存器）、`%gs`（附加段寄存器）、`%ss`（堆栈段寄存器）设置成：`INDEX=0x02`，`TI=0`、`RPL=0`，也就是段选择子为 `0x10`。

7. 准备执行 C 程序需要的环境，数据的基址为 `0x0`，栈指针从 `$start` 开始。

```
movl $0x0, %ebp
movl $start, %esp
call bootmain
```

练习 4 - 分析 bootloader 加载 ELF 格式的 OS 的过程

`waitdisk()`

这个函数等待磁盘读取操作完成，由于端口 `0x1F7` 在磁盘读取时会被设为忙状态，因此我们只需要不断地检查磁盘是否完成读写就可以完成对磁盘读取操作的等待，接下来就可以读取 `0x1F0` 端口来读取数据了。

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

`readsect()`

这个函数完成对磁盘的一个扇区的读取操作。在读取磁盘之前显然需要等待磁盘空闲。因此首先调用 `waitdisk` 函数等待之前的 `readsect` 函数的 `insl` 指令的完成。由于 bootloader 读取磁盘的方式都是 LBA 模式，相对于 CHS 模式，LBA28 模式能读取更大的磁盘空间，这就是我们选用 LBA28 模式的原因。

```
/**
 * 读取扇区编号为 secno 的数据到内存 dst 中，操作端口的方式满足 LBA28.
 * @param dst 存放磁盘数据的内存区域，由于现在是 bootloader，内存空间不需要分配即可使用
 * @param secno 要读取数据的扇区编号
 */
static void readsect(void *dst, uint32_t secno) {
```

```

waitdisk(); // 等待操作完成

outb(0x1F2, 1); // 这个函数每次只读取一个扇区的数据
outb(0x1F3, secno & 0xFF); // 扇区编号的 0~7 位
outb(0x1F4, (secno >> 8) & 0xFF); // 扇区编号的 8~15 位
outb(0x1F5, (secno >> 16) & 0xFF); // 扇区编号的 16~23 位
outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0); // 7~4 位位 1110, 表示主盘 (操作系统安装盘)
// 3~0 位为扇区编号的 24~27 位
outb(0x1F7, 0x20); // 0x20 表示读取, 0x30 表示写入

waitdisk(); // 等待操作完成
insl(0x1F0, dst, SECTSIZE / 4); // 执行读取操作
}

```

readseg()

这个函数的作用是读取从磁盘地址 offset 开始, count 字节的数据。

由于每次都读写一个扇区, 因此先计算出要读取的扇区号, 再一次性都读出来, 多余的数据不要即可。

```

static void readseg(uintptr_t va, uint32_t count, uint32_t offset) {
    uintptr_t end_va = va + count; // 内存读取的结束地址
    va -= offset % SECTSIZE; // 我们要将数据存放在 va 开始的内存区域, 但是数据读取必须扇区对齐
    // 因此我们将 va 之前向前移动到扇区开始处以便将内存区域与扇区对齐
    uint32_t secno = (offset / SECTSIZE) + 1; // 扇区号从 1 开始

    // 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++) {
        // 每个扇区都进行一次读取操作
        readsect((void *)va, secno);
    }
}

```

bootmain()

启动函数由 `bootmain.S` 汇编程序 (主引导记录) 调用, 是加载操作系统的程序, 首先读取 ELF 文件头, 根据文件头读取可执行文件的各个代码数据段的数据, 并加载进相应的内存中。

```

/* 操作系统加载入口 */
void bootmain(void) {
    // 读入磁盘的 1 号扇区 (ELF 可执行文件头地址)
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);

    // 检查魔数是否匹配, 否则当前启动盘内并没有存放 ELF 格式的可执行程序 (可启动的操作系统)
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }

    struct proghdr *ph, *eph;
}

```

```

// 根据偏移地址计算程序段头记录表
ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
eph = ph + ELFHDR->e_phnum;
for (; ph < eph; ph++) { // ph 指向程序段表的每一项
    // 由于编译后各个段（代码、常量）地址都已经被固定，因此我们从 ph 中读取出代码段的地址。
    // 由于 p_offset 是相对于文件头的偏移值（相对于 elf 本身），ELF 本身存放在 1 号扇区中，
    // 因此 ph->p_offset 就是该程序段的磁盘地址，一次性将操作系统的数据读到对应的内存中。
    readseg(ph->p_va & 0xFFFFFFFF, ph->p_memsz, ph->p_offset);
}

// 从 ELF 头中读取操作系统入口函数地址并予以调用，将 CPU 控制权转交给操作系统
// 这个函数将执行到操作系统关闭或故障
((void (*)(void))(ELFHDR->e_entry & 0xFFFFFFFF))();

bad: // 由于操作系统未能正确地写入磁盘，因此报错并等待
    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

`print_stackframe` 函数的原理很简单。首先我们需要对 C 语言函数调用进行约定：

首先堆栈指针寄存器 `esp` 指向栈的栈顶（内存布局中的最低处），比 `esp` 低的堆栈段都是可用的栈地址。向堆栈压入元素将导致 `esp` 寄存器减小，并再次指向栈顶。弹出元素时正好相反。

基址指针寄存器 `ebp` 是通用寄存器，在 C 语言程序中被用来指向函数内栈的栈底。初次进入 C 语言函数时，会向栈中压入调用者的 `ebp` 值，同时将 `ebp` 置为现在的 `esp` 值，也就是说，对于一个 C 语言函数，执行时 `ebp` 指向的地址会保存调用者的 `ebp` 地址。如果所有的函数都遵循该原则，那么我们可以利用这个约定追踪程序的调用堆栈。比如如果某个函数抛出了异常，我们可以根据调用堆栈帮助进行调试。当然，为了知道我们调用的是什么函数，我们在压入 `ebp` 的时候，还会压入当前的 `eip` 寄存器的值，也就是当前指令的地址，使我们根据这个地址找到函数定义。

如：我们调用 `printf` 的步骤如下（压入参数时需要按照顺序，因为栈是向下增长的，因此第一个参数在低地址，低地址的元素是栈的更高的元素，因此后压入栈）：

```

pushl arg2          # arg2, %ebp+4*4
pushl arg1          # arg1, %ebp+3*4
pushl format_string # arg0, %ebp+2*4
call printf
    pushl %eip # 此时的 eip 是该函数的 eip, %ebp+1*4
    pushl %ebp # 此时的 ebp 是调用者的 ebp, %ebp+0*4
    movl %esp, %ebp # 该函数的栈从当前的 esp 往下增长, 此时 ebp 指向刚才压入的 %ebp
    ...
    popl %ebp # 恢复 %ebp
    popl # 弹出 %eip
    ret
addl $12, %esp      # 弹出所有参数, 恢复栈

```

print_stackframe 函数如下:

```

void print_stackframe(void) {
    /* LAB1 YOUR CODE : STEP 1 */
    // (1) call read_ebp() to get the value of ebp. the type is (uint32_t);
    uint32_t ebp = read_ebp();
    // (2) call read_eip() to get the value of eip. the type is (uint32_t);
    uint32_t eip = read_eip();
    // (3) from 0 .. STACKFRAME_DEPTH

    int i, j;
    uint32_t *args;

    /**
     * Note that, the length of ebp-chain is limited. In boot/bootasm.S, before jumping
     * to the kernel entry, the value of ebp has been set to zero, that's the boundary.
     */
    for (i = 0; ebp && i < STACKFRAME_DEPTH; ++i) {
        // (3.1) printf value of ebp, eip
        cprintf("ebp: 0x%08x, eip: 0x%08x, args", ebp, eip);
        // (3.2) (uint32_t)calling arguments [0..4] = the contents in address (uint32_t)ebp
        +2 [0..4]
        args = (uint32_t *)ebp + 2;
        for (j = 0; j < 4; ++j) cprintf("%c 0x%08x", j == 0 ? ':' : ',', args[j]);
        // (3.3) cprintf("\n");
        cprintf("\n");
        // (3.4) call print_debuginfo(eip-1) to print the C calling function name and line
        number, etc.
        print_debuginfo(eip - 1);
        // (3.5) popup a calling stackframe
        // NOTICE: the calling funciton's return addr eip = ss:[ebp+4]
        eip = *((uint32_t *)ebp + 1); // 堆栈段寄存器值不变
        // the calling funciton's ebp = ss:[ebp]
        ebp = *((uint32_t *)ebp); // 堆栈段寄存器值不变
        //
    }
}

```

make qemu 的结果如下图所示:


```

ebp: 0x00007b38, eip: 0x00100a45, args: 0x00010094, 0x00010094, 0x00007b68, 0x00
100084
    kern/debug/kdebug.c:274: print_stackframe+21
ebp: 0x00007b48, eip: 0x00100d54, args: 0x00000000, 0x00000000, 0x00000000, 0x00
007bb8
    kern/debug/kmonitor.c:110: mon_backtrace+10
ebp: 0x00007b68, eip: 0x00100084, args: 0x00000000, 0x00007b90, 0xffff0000, 0x00
007b94
    kern/init/init.c:47: grade_backtrace2+19
ebp: 0x00007b88, eip: 0x001000a6, args: 0x00000000, 0xffff0000, 0x00007bb4, 0x00
000029
    kern/init/init.c:52: grade_backtrace1+27
ebp: 0x00007ba8, eip: 0x001000c3, args: 0x00000000, 0x00100000, 0xffff0000, 0x00
100043
    kern/init/init.c:57: grade_backtrace0+19
ebp: 0x00007bc8, eip: 0x001000e4, args: 0x00000000, 0x00000000, 0x00000000, 0x00
103520
    kern/init/init.c:62: grade_backtrace+26
ebp: 0x00007be8, eip: 0x00100050, args: 0x00000000, 0x00000000, 0x00000000, 0x00
007c4f
    kern/init/init.c:27: kern_init+79
ebp: 0x00007bf8, eip: 0x00007d6e, args: 0xc031fcfa, 0xc08ed88e, 0x64e4d08e, 0xfa
7502a8
    <unknown>: -- 0x00007d6d --

```

最下面的一行调用是 bootloader 的 bootmain 函数。bootasm.S 中：

```

# 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

```

将 %esp 设置为了 0x7c00，由于调用 C 语言函数后首先该函数会先将 eip（由 call 指令完成）和原 ebp 压入堆栈（由 C 语言完成）以便追踪调用堆栈，之后再设置 %ebp=%esp。因此最终输出的 %ebp=0x7c00-8（压了两个字进入堆栈）=0x7bf8。

至于参数，由于 args0、args1... 分别为 mem[%ebp+8]、mem[%ebp+16]、... 的值。我们考察这几个 arg 的地址：由于 %ebp=0x7bf8，所以 args0: %ebp+0x8=0x7c00、args1: %ebp+0x10=0x7c08、...。而我们知道引导程序将被加载到 0000:7c00 的内存地址，也就是说，args0、args1 等为我们的指令数据。我们可以证实一下：cli 指令的 opcode 为 0xfa、cld 指令的 opcode 为 0xfc、xorw 指令的 opcode 为 0x31（根据 <https://c9x.me/x86/> 查到），由于 x86 处理器是小端模式，低位低地址，因此从 arg0 的内容来看，证明的我们的想法是正确的：args 为引导程序代码数据。

```

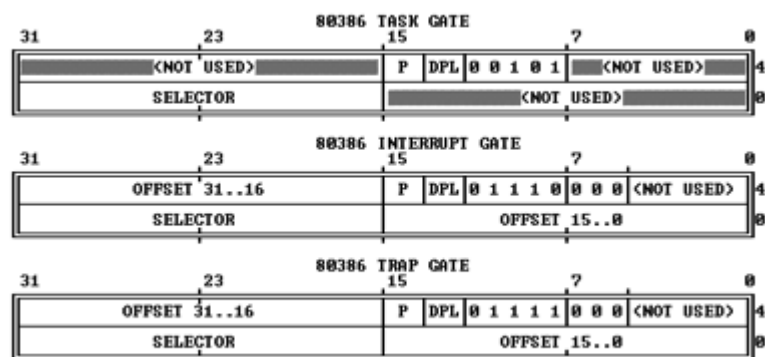
# bootasm.S 的最初的几个指令
.global start
start:
.code16                                # Assemble for 16-bit mode
    cli                                # Disable interrupts
    cld                                # String operations increment

```

练习 6 - 完善中断初始化和处理

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？

Figure 9-3. 80386 IDT Gate Descriptors



根据上图可知，一个中断描述符长度为 64 位，8 个字节。其中 **SELECTOR**、**OFFSET** 表示中断处理程序入口的地址，也就是入口地址为：**SELECTOR:OFFSET**。

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。

根据说明，我们需要依次将每个中断处理程序都通过 `SETGATE` 宏加载到 `IDT` 中。根据注意事项，我们需要将系统调用中断设置为用户态权限，其他中断均设置为内核态权限。因此设计一个 `for` 循环将所有的中断设置为内核态权限，而特别地将 `T_SWITCH_TOK` 中断（也就是 `T_SYSCALL` 中断）设置为用户态权限。最后根据提示通过 `lidt` 指令加载中断向量表。

```
void idt_init(void) {
    // LAB1 YOUR CODE : STEP 2
    // (1) where are the entry adrs of each Interrupt Service Routine (ISR)?
    // All ISR's entry adrs 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.

    // 表示各个中断处理程序的段内偏移地址
    extern uintptr_t __vectors[]; // defined in kern/trap/vector.S

    // (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

    int i;
    for (i = 0; i < IDT_SIZE; ++i) {
        // 中断处理程序在内核代码段中，特权级为内核级
        SETGATE(idt[i], GATE_INT, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }

    // 系统调用中断（陷入中断）的特权级为用户态
    SETGATE(idt[T_SWITCH_TOK], GATE_INT, GD_KTEXT, __vectors[T_SWITCH_TOK], DPL_USER);
}
```

```
// (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

lidt(&idt_pd);
}
```

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”。

根据要求，每次遇到时钟中断后更新计时器 `ticks`，每次到达 `TICK_NUM` 次，也就是 100 次后输出调试信息。

```
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
    ++ticks;
    // (2) Every TICK_NUM cycle, you can print some info using a function, such as
print_ticks().
    if (ticks == TICK_NUM) {
        ticks = 0;
        print_ticks();
    }
    // (3) Too simple? Yes, I think so!
    break;
```

```
+++ switch to user mode +++
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```

实验总结

这次实验挺复杂的，首先阅读代码就非常困难，但也因此学会了很多分析底层代码的方法。

对于练习 5：我在分析调用堆栈最底层调用的时候学会了分析内存的方法。我首先彻底地了解了 C 函数的调用过程，知道了进入函数时会将 eip、ebp 压栈，分析得知了 print_stackframe 函数内要求我们做的事情的原理，因此也知道了 args 不过也只是栈中的数据，而栈只是我们人为规定的内存空间而已。因此再计算出这些 args 的内存地址就知道了这些都其实在代码段（因为我们将 0~7BFF 的内存挪作栈使用了）。也就分析出了 args 的数据的含义。

通过这次试验，我知道了如何使用 gdb 进行调试，了解了主引导程序的原理、操作系统的启动过程、操作系统的中断控制方法、内核态用户态的切换。同时我对练习 5 的印象最深刻，因为这解释了一些程序设计理论题的原理，解决了我遇到的一些疑难问题。