Lab1

前言

这部分的题头是“启动PC”，顾名思义是引导加载程序的使用。第一部分是配置环境，已经组合在lab0中，接下来就是gdb调试。

使用gdb调试

首先使用gdb，这需要两个Terminal。

```
/mnt/c/Users/95225$ cd ~/lab
~/lab$ make qemu-nox-gdb
```

第二个Terminal输入：

```
/mnt/c/Users/95225$ cd ~/lab
~/lab$ make gdb
```

如果产生了这样的内容就说明进入了

```
The target architecture is assumed to be i8086
[f000:fff0]  0xffff0: ljmp  $0xf000,$0xe05b
0x0000fff0 in ?? ()
```

Lab1的exercise2

问题

使用 `si` 查看正在进行的ROM BIOS指令，看看这些指令是在干什么的?

解答

首先看看这些指令是什么东西：

```
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
[f000:e05b]    0xfe05b: cmpl   $0x0,%cs:0x6ac8
[f000:e062]    0xfe062: jne    0xfd2e1
[f000:e066]    0xfe066: xor    %dx,%dx
[f000:e068]    0xfe068: mov    %dx,%ss
[f000:e06a]    0xfe06a: mov    $0x7000,%esp
[f000:e070]    0xfe070: mov    $0xf34c2,%edx
[f000:e076]    0xfe076: jmp    0xfd15c
[f000:d15c]    0xfd15c: mov    %eax,%ecx
[f000:d15f]    0xfd15f: cli
[f000:d160]    0xfd160: cld
[f000:d161]    0xfd161: mov    $0x8f,%eax
[f000:d167]    0xfd167: out    %al,$0x70
[f000:d169]    0xfd169: in     $0x71,%al
```

```
[f000:d16b]    0xfd16b: in     $0x92,%al
[f000:d16d]    0xfd16d: or     $0x2,%al
[f000:d16f]    0xfd16f: out    %al,$0x92
[f000:d171]    0xfd171: lidtw  %cs:0x6ab8
[f000:d177]    0xfd177: lgdtw  %cs:0x6a74
[f000:d17d]    0xfd17d: mov    %cr0,%eax
[f000:d180]    0xfd180: or     $0x1,%eax
[f000:d184]    0xfd184: mov    %eax,%cr0
[f000:d187]    0xfd187: ljmpl  $0x8,$0xfd18f
The target architecture is assumed to be i386
=> 0xfd18f:     mov    $0x10,%eax
0x000fd18f in ?? ()
```

到最后一步发生一个长跳转，进入保护模式，实模式结束。

大部分人对于这一个exercise直接跳过，这里还是要给出一个大致的分析的，但是由于这部分还是属于对于底层控制要求非常高，至少要读过很多标准。

但是大概可以看出这个在做什么。

1. 在`cli`处禁止中断，这样可以防止在修改`ss`和`sp`之间插入代码的时候发生中断导致跳转到不知道哪里去了。
2. 之后是`cld`，设置了从低地址到高地址的复制方式。
3. 和IO设备交互(`0xfd167`~`0xfd16b`)的同时打开a20门。
4. 加载`lidtw`和`lgdtw`。
5. `Enable %cr0`，进入实模式。之后的地址（包括`ljmpl`这一行）都是保护模式进行。

Lab1的exercise3

问题

在地址`0x7c00`设置断点，这是引导扇区将被加载的位置。继续执行直到那个断点。跟踪`boot/boot.S`中的代码，使用源代码和反汇编文件`obj/boot/boot.asm`来跟踪你在哪里。还可以使用GDB中的`x/i`命令来反汇编引导加载程序中的指令序列，并将原始引导加载程序源代码与`obj/boot/boot.asm`和GDB中的反汇编进行比较。

在`boot/main.c`中跟踪到`bootmain()`，然后进入`readsect()`。识别与`readsect()`中的每个语句相对应的确切的汇编指令。完成对`readsect()`其余部分跟踪，并回到`bootmain()`，并找到用来从盘读取内核其余部分的`for`循环的开始和结束的位置。找出循环完成后运行的代码，在那里设置一个断点，并继续执行到该断点。然后再走完bootloader程序的其余部分。

确保你可以回答以下问题：

1. 处理器什么时候开始执行32位代码？如何完成的从16位到32位模式的切换？
2. 引导加载程序bootloader执行的最后一个指令是什么，加载的内核的第一个指令是什么？
3. 内核的第一个指令在哪里？
4. 引导加载程序如何决定为了从磁盘获取整个内核必须读取多少扇区？在哪里可以找到这些信息？

前置知识

和其他地方不一样的是，这里会讲解的是解答中不应该提及但是对理解有帮助的内容。这些内容可以引导出更多的知识，也方便对比我们熟悉的系统和OS的区别。

调试需要的gdb命令

调试可以使用下面的gdb指令。注意，在网上查找到的不是所有的gdb指令可以用，因为那是基于操作系统实现的，但是在lab1中，操作系统并没有启动。这里的第一条指令是设置断点，第二条是查看寄存器，后续会涉及到更多用法。

```
(gdb) b *0x7c00
(gdb) i registers
```

硬盘端口

端口号	功能
1F0	数据寄存器，读写数据都从这里走
1F1	逻辑寄存器，每一位表示一个错误，全零表示成功
1F2	扇区计数，存放需要操作的扇区数量
1F3~1F5	扇区LBA地址的0-23位
1F6	低四位表示LBA地址的24-27位，第四位如果为0则是主盘，反之从盘，其他必须为1
1F7	写的时候就是命令寄存器，0x20可以推测为写入。读的时候返回状态。

ELF结构体

注意这里的ELF和实际linux中的ELF是非常相似的。

```
#define ELF_MAGIC 0x464C457FU // "\x7FELF" in little endian
struct elfhdr
{
    uint magic; //4字节，0x464C457FU（大端模式）或 0x7felf（小端模式）
    uchar elf[12]; // 12 字节，每字节对应意义如下：
    //      0 : 1 = 32 位程序；2 = 64 位程序
    //      1 : 数据编码方式，0 = 无效；1 = 小端模式；2 = 大端模式
    //      2 : 只是版本，固定为 0x1
    //      3 : 目标操作系统架构
    //      4 : 目标操作系统版本
    //      5 ~ 11 : 固定为 0
    ushort type; // 2 字节，表明该文件类型，意义如下：
    //      0x0 : 未知目标文件格式
    //      0x1 : 可重定位文件
    //      0x2 : 可执行文件
    //      0x3 : 共享目标文件
    //      0x4 : 转储文件
    //      0xff00 : 特定处理器文件
    //      0xffff : 特定处理器文件
    ushort machine;  //这里我们只需要知道 0x0 为未指定；0x3 为 x86 架构
    uint version;    //4字节，表示该文件的版本号
    uint entry;      //4字节，该文件的入口地址，非可执行文件则为 0
    uint phoff;      //4字节，表示该文件“程序头部表”相对位置，单位是字节
    uint shoff;      //4字节，表示该文件“节区头部表”相对位置，单位是字节
    uint flags;      //4字节，特定处理器标志
    ushort ehsize;   //2字节，ELF文件头部的大小，单位是字节
    ushort phentsize; //2字节，程序头部表中一个入口的大小，单位是字节
    ushort phnum;   // 2 字节，表示程序头部表的入口个数，
    //phnum * phentsize = 程序头部表大小（单位是字节）
```

```
    ushort shentsize; // 2 字节，节区头部表入口大小，单位是字节
    ushort shnum;   // 2 字节，节区头部表入口个数，
    // shnum * shentsize = 节区头部表大小（单位是字节）
    ushort shstrndx;  // 2 字节，表示字符表相关入口的节区头部表索引
};
// 程序头表
struct proghdr
{
    uint type;   // 4 字节， 段类型
    // 1 PT_LOAD : 可载入的段
    // 2 PT_DYNAMIC : 动态链接信息
    // 3 PT_INTERP : 指定要作为解释程序调用的以空字符结尾的路径名的位置和大小
    // 4 PT_NOTE : 指定辅助信息的位置和大小
    // 5 PT_SHLIB : 保留类型，但具有未指定的语义
    // 6 PT_PHDR : 指定程序头表在文件及程序内存映像中的位置和大小
    // 7 PT_TLS : 指定线程局部存储模板
    uint off;    //4字节，段的第一个字节在文件中的偏移
    uint vaddr;  //4字节，段的第一个字节在内存中的虚拟地址
    uint paddr;  //4字节，段的第一个字节在内存中的物理地址
    uint filesz; // 4 字节， 段在文件中的长度
    uint memsz;  // 4 字节， 段在内存中的长度
    uint flags;  // 4 字节， 段标志
    //         1 : 可执行
    //         2 : 可写入
    //         4 : 可读取
    uint align;  // 4 字节， 段在文件及内存中如何对齐
};
```

下面是在`inc/elf.h`中的`elf`结构体

```
#define ELF_MAGIC 0x464C457FU /* "\x7FELF" in little endian */
struct Elf
{
    uint32_t e_magic; // must equal ELF_MAGIC
    uint8_t e_elf[12];
    uint16_t e_type;
    uint16_t e_machine;
    uint32_t e_version;
    uint32_t e_entry;
    uint32_t e_phoff;
    uint32_t e_shoff;
    uint32_t e_flags;
    uint16_t e_ehsize;
    uint16_t e_phentsize;
    uint16_t e_phnum;
    uint16_t e_shentsize;
    uint16_t e_shnum;
    uint16_t e_shstrndx;
};
struct Proghdr
{
    uint32_t p_type;
    uint32_t p_offset;
    uint32_t p_va;
    uint32_t p_pa;
    uint32_t p_filesz;
    uint32_t p_memsz;
```

```
    uint32_t p_flags;
    uint32_t p_align;
};
```

解答

既然问题很多，就从问题入手。

1. 处理器什么时候开始执行32位代码？如何完成的从16位到32位模式的切换？

   首先进入调试，在`0x7c00`打上断点，然后不断`si`，直到看到i386字样出现，这代表进入了80386的模式。对照`obj/boot/boot.asm`可以发现下列指令：

```
  .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
  # 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
  testb   $0x2,%al
  jnz     seta20.1
  movb    $0xd1,%al               # 0xd1 -> port 0x64
  outb    %al,$0x64
seta20.2:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2
  movb    $0xdf,%al               # 0xdf -> port 0x60
  outb    %al,$0x60
  # Switch from real to protected mode, using a bootstrap GDT
  # and segment translation that makes virtual addresses
  # identical to their physical addresses, so that the
  # effective memory map does not change during the switch.
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE_ON, %eax
  movl    %eax, %cr0
  # Jump to next instruction, but in 32-bit code segment.
  # Switches processor into 32-bit mode.
  ljmp    $PROT_MODE_CSEG, $protcseg
```

   很容易就可以知道是最后几行完成了16位到32位的转换。整个部分的执行过程是初始化段寄存器→打开A20门→设置GDT和cr0寄存器→跳转到虚模式。

   GDT是全局描述符表，对应的`GDTR`是存储这个表在哪里的寄存器，想要在保护模式下进行内存寻址首先就要有GDT，不然查不到正确的表。这东西在本lab还有戏份。A20门主要用于打开1MB的内存限制，防止CPU被卷绕机制卡住，这个问题主要由IBM提出和解决，可以考虑查一下计算机历史验证一下。

2. 引导加载程序bootloader执行的最后一个指令是什么，加载的内核的第一个指令是什么？

和之前一样设置断点进入`bootmain`，然后继续`si`，不过这里还提供了另外一条路，就是对照C代码来阅读，除了不能F12跳转之外其他并没有什么差异。

在`boot/main.c`可以找到`bootmain`的代码，理论上这段代码进行结束以后就要跳转到内核，并且如果不存在错误的话，应当就退到这一行。直接找到对应的asm代码，在`obj/boot/boot.asm`中可以看到

```
((void (*)(void)) (ELFHDR->e_entry))();
7d6b:   ff 15 18 00 01 00       call   *0x10018
```

这就是内核的最后一条指令。

3. 内核的第一个指令在哪里?

   感觉是重复问题。

```
(gdb) si
=> 0x10000c:    movw   $0x1234,0x472
0x0010000c in ?? ()
```

4. 引导加载程序如何决定为了从磁盘获取整个内核必须读取多少扇区？在哪里可以找到这些信息?

   这一部分的答案是不确定的，主要取决于这一部分。

```
for (; ph < eph; ph++)
// p_pa is the load address of this segment (as well as the physical address)
    readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
```

接下来接着第二题继续分析`bootmain`在干什么，首先阅读`boot/main.c`中的`bootmain`函数。

```
void bootmain(void)
{
    struct Proghdr *ph, *eph;
    // read 1st page off disk
    // 读第一个区块，这里和MBR分区非常相似
    readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);
    if (ELFHDR->e_magic != ELF_MAGIC)
        goto bad;
    // load each program segment (ignores ph flags)
    ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
    // 按顺序读各个程序块（这里没有分区表）
    for (; ph < eph; ph++)
        readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
    // call the entry point from the ELF header
    ((void (*)(void)) (ELFHDR->e_entry))();
bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);
    while (1);
}
```

这一块的大致意思是，首先读取第一个扇区(512bytes)的内容，如果阅读到的内容是我们需要的常量，那么继续，否则报错。这个常量在前面的ELF中可以找到。接下来根据第一个扇区读取其他的内容。原理是类似的。读取完毕就进入内核。

接下来看下`readseg()`在干什么，同样的文件中阅读可以得知。

```
#define SECTSIZE 512
void readseg(uint32_t pa, uint32_t count, uint32_t offset)
{
    uint32_t end_pa;
    end_pa = pa + count;
    // 512字节对齐
    pa &= ~(SECTSIZE - 1);
    // 注意这里是从1开始的
    offset = (offset / SECTSIZE) + 1;
    while (pa < end_pa) {
        readsect((uint8_t*) pa, offset);
        pa += SECTSIZE;
        offset++;
    }
}
```

这个函数中在做读取什么东西的操作，首先进行了512字节的对齐（从这里就可以猜测是MBR了，因为几乎只有MBR的头512字节是进入内核的必要元素）；然后算了下偏移，因为这里不只是第一个扇区要读，还有其他的内容需要进行读取；接着就是读取，涉及到了`readsect`函数，似乎是512字节一块这样读取，在循环中一同移动`offset`。

接着就是`readsect`。

```
void waitdisk(void) { while ((inb(0x1F7) & 0xC0) != 0x40); }
void readsect(void *dst, uint32_t offset)
{
    // 等待磁盘
    waitdisk();
    outb(0x1F2, 1);     // count = 1
    // 这四个字节的读法自行对照表格
    outb(0x1F3, offset);
    outb(0x1F4, offset >> 8);
    outb(0x1F5, offset >> 16);
    outb(0x1F6, (offset >> 24) | 0xE0);
    outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
    // 磁盘可能一下反应不过来
    waitdisk();
    insl(0x1F0, dst, SECTSIZE/4);
}
```

这里首先等待磁盘，然后不停地输入，接着又等了磁盘，最后就是拿取输出。看上去一堆IO交互，实际上看一下前面的内容就可以知道在干什么。

至于`inb/outb/insl`这些函数，进入`inc/x86.h`就可以知道全部答案。不过这里没有解释为什么叫这些函数，因为inb = input byte，outb = output byte，insl = input string long（这个l是啥缩写还真不知道，之前的猜测四个字节应该没有错，但那应该是qword）。

```
static inline uint8_t inb(int port)
{
    uint8_t data;
    asm volatile("inb %w1,%0" : "=a" (data) : "d" (port));
    return data;
}
static inline void insl(int port, void *addr, int cnt)
```

```
{
    asm volatile("cld\n\trepne\n\tinsl"
                 : "=D" (addr), "=c" (cnt)
                 : "d" (port), "0" (addr), "1" (cnt)
                 : "memory", "cc");
}
static inline void outb(int port, uint8_t data)
{
    asm volatile("outb %0,%w1" : : "a" (data), "d" (port));
}
```

Lab1的exercise4

问题

1. 阅读指针编程的相关书籍。
2. 确定以下代码的输出。

```
void f(void)
{
    int a[4];
    int *b = malloc(16);
    int *c;
    int i;
    printf("1: a = %p, b = %p, c = %p\n", a, b, c);
    c = a;
    for (i = 0; i < 4; i++)
    a[i] = 100 + i;
    c[0] = 200;
    printf("2: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
       a[0], a[1], a[2], a[3]);

    c[1] = 300;
    *(c + 2) = 301;
    3[c] = 302;
    printf("3: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
       a[0], a[1], a[2], a[3]);

    c = c + 1;
    *c = 400;
    printf("4: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
       a[0], a[1], a[2], a[3]);

    c = (int *) ((char *) c + 1);
    *c = 500;
    printf("5: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
       a[0], a[1], a[2], a[3]);

    b = (int *) a + 1;
    c = (int *) ((char *) a + 1);
    printf("6: a = %p, b = %p, c = %p\n", a, b, c);
}
```

解答

运行代码可以得到如下输出：

```
1: a = 0x7fffccc0fa60, b = 0x7fffc5f7c260, c = 0x7fe38f2008fd
2: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103
3: a[0] = 200, a[1] = 300, a[2] = 301, a[3] = 302
4: a[0] = 200, a[1] = 400, a[2] = 301, a[3] = 302
5: a[0] = 200, a[1] = 128144, a[2] = 256, a[3] = 302
6: a = 0x7fffccc0fa60, b = 0x7fffccc0fa64, c = 0x7fffccc0fa61
```

实际上这里的难点主要是第一行。后面的都是简单的指针。实际上把第一行的abc的输出加入一个&取出其地址，可以得到如下的结果：

```
1: a = 0x7fffebc98db0, b = 0x7fffebc98da0, c = 0x7fffebc98da8
```

这个时候可能就很明显了，这才是作者想要阐释的道理，就是局部内存是定义在栈上面的，并且是从高往低进展的。

Lab1的exercise5

问题

根据bootloader的代码找到修改链接地址可以引发错误的地方以后进行修改，观察运行情况。

解答

这里还是不找了，直接把`boot/Makeflag`的入口`7c00`改成`8c00`，然后重新编译，发现得到了这样的输出。

```
/usr/local/qemu/bin/qemu-system-i386 -nographic -drive
file=obj/kern/kernel.img,index=0,media=disk
,format=raw -serial mon:stdio -gdb tcp::26000 -D qemu.log  -S
EAX=00000011 EBX=00000000 ECX=00000000 EDX=00000080
ESI=00000000 EDI=00000000 EBP=00000000 ESP=00006f20
EIP=00007c2d EFL=00000006 [-----P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0000 00000000 0000ffff 00009300 DPL=0 DS16 [-WA]
CS =0000 00000000 0000ffff 00009b00 DPL=0 CS16 [-RA]
SS =0000 00000000 0000ffff 00009300 DPL=0 DS16 [-WA]
DS =0000 00000000 0000ffff 00009300 DPL=0 DS16 [-WA]
FS =0000 00000000 0000ffff 00009300 DPL=0 DS16 [-WA]
GS =0000 00000000 0000ffff 00009300 DPL=0 DS16 [-WA]
LDT=0000 00000000 0000ffff 00008200 DPL=0 LDT
TR =0000 00000000 0000ffff 00008b00 DPL=0 TSS32-busy
GDT=     00000000 00000000
IDT=     00000000 000003ff
CR0=00000011 CR2=00000000 CR3=00000000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
EFER=0000000000000000
Triple fault.  Halting for inspection via QEMU monitor.
```

可以知道在进入bootloader以后还是可以得到正常的输出，之后才有问题。而且除此以外，入口似乎还是`7c00`，这是因为BIOS还是没有得到修改。重新编译以后以后然后重新进入gdb看以下情况。发现了这两个错误。

```
[   0:7c1e] => 0x7c1e:  lgdtw  -0x739c
[   0:7c2d] => 0x7c2d:  ljmp   $0x8,$0x8c32
```

引出的问题

你真的以为这样就结束了吗？实际上并没有，我们只是挑选了其中一个比较简单的情况来处理这个题目：负数内存肯定比正数不好读取。而且除此以外我们也没有解释这个`-0x739c`内存是从哪里来的。

这里感谢@black_desk给出的解释。把附近的字节码拿出来看看。

```
(gdb) x/64b 0x7c1e
0x7c1e: 0x0f    0x01    0x16    0x64    0x8c    0x0f    0x20    0xc0
0x7c26: 0x66    0x83    0xc8    0x01    0x0f    0x22    0xc0    0xea
0x7c2e: 0x32    0x8c    0x08    0x00    0x66    0xb8    0x10    0x00
0x7c36: 0x8e    0xd8    0x8e    0xc0    0x8e    0xe0    0x8e    0xe8
0x7c3e: 0x8e    0xd0    0xbc    0x00    0x8c    0x00    0x00    0xe8
0x7c46: 0xcb    0x00    0x00    0x00    0xeb    0xfe    0x00    0x00
0x7c4e: 0x00    0x00    0x00    0x00    0x00    0x00    0xff    0xff
0x7c56: 0x00    0x00    0x00    0x9a    0xcf    0x00    0xff    0xff
```

因为`8c64`的补码就是`-0x739c`，所以就是这样。

走过了这一个问题，又产生了下一个问题：如果是`0x6c00`呢？我们重新用`0x6c00`调试一下，发现这个结果：

```
(gdb) x/64b 0x7c1e
0x7c1e: 0x0f    0x01    0x16    0x64    0x6c    0x0f    0x20    0xc0
0x7c26: 0x66    0x83    0xc8    0x01    0x0f    0x22    0xc0    0xea
0x7c2e: 0x32    0x6c    0x08    0x00    0x66    0xb8    0x10    0x00
0x7c36: 0x8e    0xd8    0x8e    0xc0    0x8e    0xe0    0x8e    0xe8
0x7c3e: 0x8e    0xd0    0xbc    0x00    0x6c    0x00    0x00    0xe8
0x7c46: 0xcb    0x00    0x00    0x00    0xeb    0xfe    0x00    0x00
0x7c4e: 0x00    0x00    0x00    0x00    0x00    0x00    0xff    0xff
0x7c56: 0x00    0x00    0x00    0x9a    0xcf    0x00    0xff    0xff
```

于是乎就有这些数值：

```
(gdb) x/64b 0x6c64
0x6c64: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x6c6c: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x6c74: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x6c7c: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x6c84: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x6c8c: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x6c94: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x6c9c: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) x/64b 0x7c64
0x7c64: 0x17    0x00    0x4c    0x6c    0x00    0x00    0x55    0xba
0x7c6c: 0xf7    0x01    0x00    0x00    0x89    0xe5    0xec    0x83
0x7c74: 0xe0    0xc0    0x3c    0x40    0x75    0xf8    0x5d    0xc3
0x7c7c: 0x55    0x89    0xe5    0x57    0x8b    0x4d    0x0c    0xe8
0x7c84: 0xe2    0xff    0xff    0xff    0xb0    0x01    0xba    0xf2
0x7c8c: 0x01    0x00    0x00    0xee    0xba    0xf3    0x01    0x00
0x7c94: 0x00    0x88    0xc8    0xee    0x89    0xc8    0xba    0xf4
0x7c9c: 0x01    0x00    0x00    0xc1    0xe8    0x08    0xee    0x89
```

```
(gdb) x/64b 0x6c32
0x6c32: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x6c3a: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x6c42: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x6c4a: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x6c52: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x6c5a: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x6c62: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x6c6a: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

GDT全0，`6c64`能正常解码，但是跳转的位置全0，于是乎产生了死循环，由于`ljmp`没有这个行为，因此猜测是QEMU的行为。

Lab1的exercise6

问题

复位机器（退出QEMU/GDB并再次启动）。在BIOS进入引导加载程序的那一刻停下来，检查内存中`0x00100000`地址开始的8个字的内容，然后再次运行，到bootloader进入内核的那一点再停下来，再次打印内存`0x00100000`的内容。为什么这8个字的内容会有所不同？第二次停下来的时候，打印出来的内容是什么？（你不需要使用QEMU来回答这个问题，思考即可）

前置知识

MBR结构，取自于wiki。

十六进制	描述	长度
0x000	代码区	440（最大446）
0x1B8	选用磁盘标志	4
0x1BC	一般为0x0000	2
0x1BE	MBR分区规划，四个16byte的主分区表入口	64
0x1FE	0x55AA	2

解答

这两个显然是不一样的，因为在最后内核加载进来了。

```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli
Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/16x 0x00100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
0x100020:       0x00000000      0x00000000      0x00000000      0x00000000
0x100030:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) b *0x7d6b
Breakpoint 2 at 0x7d6b
```

```
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d6b:      call   *0x10018
Breakpoint 2, 0x00007d6b in ?? ()
(gdb) si
=> 0x10000c:    movw   $0x1234,0x472
0x0010000c in ?? ()
(gdb) x/16x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x2000b812      0x220f0011      0xc0200fd8
0x100020:       0x0100010d      0xc0220f80      0x10002fb8      0xbde0fff0
0x100030:       0x00000000      0x110000bc      0x0068e8f0      0xfeeb0000
```

在哪里读取的？肯定是`readsect`，具体是哪一个建议重新开坑。

引出的问题

之前的实验中没有体现出`readsect`读到哪里去了，实际上是可以找到的，因为第一个MBR肯定存储了更多的程序段。这里又回到了在之前提到的坑：`bootmain`循环读取的内容到底是什么东西。之前的猜测是读取的分区，但是后续的证明把它否定了。

Lab1的exercise7

问题

使用QEMU和GDB跟踪到JOS内核并停止在`movl %eax, %cr0`。查看内存中在地址`0x00100000`和`0xf0100000`处的内容。下面，使用GDB命令`stepi`单步执行该指令。指令执行后，再次检查`0x00100000`和`0xf0100000`的内存。确保你明白刚刚发生的事情。

新映射建立后的第一条指令是什么，如果映射配置错误，它还能不能正常工作？注释掉`kern/entry.S`中的`movl %eax, %cr0`，再次追踪到它，看看你的猜测是否正确。

前置知识-A20门

8086能找到的最大内存为：`FFFFh : FFFFh= 10FFEFh = 1M+64K-16Bytes`（1M多余出来的部分被称做高端内存区HMA）。但8086/8088只有20位地址线，只能够访问1M地址范围的数据，所以如果访问`0x100000`~`0x10FFEF`之间的内存（大于1M空间），则必须有第21根地址线来参与寻址（8086/8088没有）。因此，当程序员给出超过1M的地址时，系统计算实际地址按照对1M求模的方式进行的，这种技术被称为**wrap-around**。

对于80286或以上的CPU通过A20GATE来控制A20地址线。技术发展到了80286，虽然系统的地址总线由原来的20根发展为24根，这样能够访问的内存可以达到2^24=16M，但是Intel在设计80286时提出的目标是向下兼容,所以在实模式下，系统所表现的行为应该和8086/8088所表现的完全一样，也就是说，在实模式下，80386以及后续系列应该和8086/8088完全兼容仍然使用A20地址线。所以说80286芯片存在一个BUG：它开设A20地址线。如果程序员访问`0x100000H`-`0x10FFEF`之间的内存，系统将实际访问这块内存（没有wrap-around技术），而不是像8086/8088一样从0开始。

注意在`boot/boot.S`中没有涉及到CR0_PG，但是在`kern/entry.S`中找到了这一个东西，也就是说asm代码是存在问题的。

解答

解答以上问题可能需要阅读非常多的材料，甚至可以再开一个小型的实验

首先尝试操作一下得到如下结果（不动代码）。其中entry.S的那条语句在kernel.asm里面可以找到，地址为0xf010025，把根据题目提示可以知道映射到0x100025。

```
(gdb) b *0x100025
Breakpoint 1 at 0x100025
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x100025:    mov    %eax,%cr0
Breakpoint 1, 0x00100025 in ?? ()
(gdb) x/64b 0x100000
0x100000: 0x02 0xb0 0xad 0x1b 0x00 0x00 0x00 0x00
0x100008: 0xfe 0x4f 0x52 0xe4 0x66 0xc7 0x05 0x72
0x100010: 0x04 0x00 0x00 0x34 0x12 0xb8 0x00 0x20
0x100018: 0x11 0x00 0x0f 0x22 0xd8 0x0f 0x20 0xc0
0x100020: 0x0d 0x01 0x00 0x01 0x80 0x0f 0x22 0xc0
0x100028: 0xb8 0x2f 0x00 0x10 0xf0 0xff 0xe0 0xbd
0x100030: 0x00 0x00 0x00 0x00 0xbc 0x00 0x00 0x11
0x100038: 0xf0 0xe8 0x68 0x00 0x00 0x00 0xeb 0xfe
(gdb) x/64b 0xf0100000
0xf0100000 <_start+4026531828>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xf0100008 <_start+4026531836>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xf0100010 <entry+4>:  0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xf0100018 <entry+12>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xf0100020 <entry+20>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xf0100028 <entry+28>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xf0100030 <relocated+1>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xf0100038 <relocated+9>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
(gdb) si
=> 0x100028: mov $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) x/64b 0xf0100000
0xf0100000 <_start+4026531828>: 0x02 0xb0 0xad 0x1b 0x00 0x00 0x00 0x00
0xf0100008 <_start+4026531836>: 0xfe 0x4f 0x52 0xe4 0x66 0xc7 0x05 0x72
0xf0100010 <entry+4>:   0x04 0x00  0x00 0x34 0x12 0xb8 0x00 0x20
0xf0100018 <entry+12>:  0x11 0x00 0x0f 0x22 0xd8 0x0f 0x20 0xc0
0xf0100020 <entry+20>:  0x0d 0x01 0x00 0x01 0x80 0x0f 0x22 0xc0
0xf0100028 <entry+28>:  0xb8 0x2f 0x00 0x10 0xf0 0xff 0xe0 0xbd
0xf0100030 <relocated+1>:    0x00 0x00 0x00 0x00 0xbc 0x00 0x00 0x11
0xf0100038 <relocated+9>:    0xf0 0xe8 0x68 0x00 0x00 0x00 0xeb 0xfe
```

可以看出在保护模式打开了以后，产生了一个映射。

然后把要求的一行注释掉，重新调试，得到这样的结果：

```
(gdb) b *0x100025
Breakpoint 1 at 0x100025
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x100025:    mov    $0xf010002c,%eax
Breakpoint 1, 0x00100025 in ?? ()
(gdb) si
=> 0x10002a:    jmp    *%eax
0x0010002a in ?? ()
(gdb) si
=> 0xf010002c <relocated>:      add    %al,(%eax)
```

```
relocated () at kern/entry.S:74
74              movl    $0x0,%ebp                       # nuke frame
pointer
(gdb)
Remote connection closed
```

引出的问题

在刚才的实验中，我们可以观察到虚拟地址的建立，但是这个地址是如何被转化成需要的地址的?

Lab1的exercise8

问题

解答以上问题可能需要阅读非常多的材料，甚至可以再开一个小型的实验

我们省略了一小段代码-使用%o形式的模式打印八进制数字所需的代码。查找并补全这个代码片段。

确保你能回答以下问题（具体内容见回答问题部分）。

前置知识

C语言不知可以传递定长的参数，也可以和C++一样传输`va_arg`，具体内容则是`va_list`，在C++中则依靠模板和`...`这个语法特性来完成。

解答

代码练习有样学样。在`lib/printfmt.c`中找到这一段代码：

```
case 'o':
// Replace this with your code.
    putch('X', putdat);
    putch('X', putdat);
    putch('X', putdat);
    break;
```

换成下面的就行。

```
case 'o':
    num = getuint(&ap, lflag);
    base = 8;
    goto number;
```

接下来回答问题：

1. 解释`printf.c`和`console.c`之间的接口。具体来说，`console.c`导出了什么函数?`printf.c`是如何使用这些函数的?

   这里主要是`printf.c`调用了`console.c`，`console.c`调用了系统调用。更具体地说是`cprintf -> vcprintf -> (printfmt.c) -> putch(printf.c) -> cputchar -> cputchar(console.c)`。

2. 解释代码：

```
if (crt_pos >= CRT_SIZE)
{
    int i;
    memmove(crt_buf, crt_buf + CRT_COLS,
            (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
    for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
    crt_buf[i] = 0x0700 | ' ';
    crt_pos -= CRT_COLS;
}
```

这一部分的意思是说，如果crt的光标位置大于CRT的容纳面积，那么从crt的缓冲区移走最上面一行代码，然后最后一行用黑色填满。

3. 跟踪以下代码的并单步执行。

```
int x = 1, y = 3, z = 4;
cprintf("x %d, y %x, z %d\n", x, y, z);
```

首先进入`vcprintf`，其中`fmt`指向第一个字符数组，`ap`指向`x`的地址。之后的过程实在过于冗长而且无聊，加上找到这个添加代码的地方不是很容易，这题被迫选择跳过，如果想看答案的或者人肉调试的，自己去这个链接看。

4. 运行下面的代码。

```
unsigned int i = 0x00646c72;
cprintf("H%x Wo%s", 57616, &i);
```

MIT你这是在玩梗吗？拿1当l用来输出hello world。

5. 在下面的代码中，将在y=之后打印什么（注意：答案不是一个固定的值）？为什么会发生这种情况?

```
cprintf("x=%d y=%d", 3);
```

这种情况下，打印出来的值就是存储x后面四个字节代表的值，因为打印出x之后，`va_arg`的`ap`指针会指向x的下一个字节，因此不管调用几个参数，在解析到%d的时候，`va_arg`就会把当前指针的地址作为一个`int`整数返回。

6. 假设GCC更改了它的调用约定，以声明的顺序将参数压入栈中，这样会使最后一个参数最后被压入。你将如何更改`cprintf`或其接口，以便仍然可以传递一个可变数量的参数?

解决方法有两个：一个是入乡随俗，把处理顺序改成和GCC相同，但是可能可读性很差。另一个是在原接口的最后加入一个`int`，用来记录所有参数的总长度，然后进入尾递归。

但是都很麻烦啊，所以不如改回来吧（笑）。

Lab1的exercise9

问题

确定内核在哪里完成了栈的初始化，以及栈所在内存的确切位置。内核如何为栈保留空间？栈指针初始化时指向的是保留区域的哪一端?

解答

在`boot/kern/kernel.asm`里面直接搜索`$esp`，看看什么时候赋上了值。然后尝试打断点进去，发现无法进去。只能通过间接的方法，从附近(最近实验的cr0)进去看看，发现了这些东西。

```
(gdb) b *0x100020
Breakpoint 1 at 0x100020
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x100020:    or     $0x80010001,%eax
Breakpoint 1, 0x00100020 in ?? ()
(gdb) si
=> 0x100025:    mov    %eax,%cr0
0x00100025 in ?? ()
(gdb)
=> 0x100028:    mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb)
=> 0x10002d:    jmp    *%eax
0x0010002d in ?? ()
(gdb)
=> 0xf010002f <relocated>:      mov    $0x0,%ebp
relocated () at kern/entry.S:74
74              movl    $0x0,%ebp                       # nuke frame
pointer
(gdb)
=> 0xf0100034 <relocated+5>:    mov    $0xf0110000,%esp
relocated () at kern/entry.S:77
77              movl    $(bootstacktop),%esp
```

可以看出发生了地址的突变，可能是GDT跳转的问题。从最后可以看出就是这里给esp赋上的值，大小是`0xf00110000`。根据已经有的知识，栈应该从高地址往低地址走，所以这里应该定义的是高地址。

Lab1的exercise10

问题

熟悉x86上C语言函数的调用约定，在`obj/kern/kernel.asm`中找到`test_backtrace`函数的地址，在其中设置一个断点，并检查在内核启动后每次这个函数被调用时会发生什么。每一级的`test_backtrace`在递归调用时，会在栈上压入多少个32位的字，这些字的内容是什么?

请注意，为了使此练习正常工作，你应该使用工具页中修改过QEMU。否则，你必须手动将所有断点和内存地址转换为线性地址。

前置知识

1. 严重bug

   注意这里可能会有一个问题，就是在打上b断点以后会发现

   ```
   (gdb) b test_backtrace
   warning: Breakpoint address adjusted from 0xf0100040 to
   0xffffffffff0100040.
   Breakpoint 1 at 0xf0100040: file kern/init.c, line 13.
   (gdb) c
   Continuing.

   ```

```
Program received signal SIGTRAP, Trace/breakpoint trap.
The target architecture is assumed to be i386
=> 0xf0100040 <test_backtrace>: push   %ebp
test_backtrace (x=5) at kern/init.c:13
13      {
(gdb) si
=> 0xf0100040 <test_backtrace>: push   %ebp
13      {
(gdb)
=> 0xf0100040 <test_backtrace>: push   %ebp
13      {
(gdb)
=> 0xf0100040 <test_backtrace>: push   %ebp
13      {
(gdb)
=> 0xf0100040 <test_backtrace>: push   %ebp
13      {
(gdb)
=> 0xf0100040 <test_backtrace>: push   %ebp
13      {
(gdb)
=> 0xf0100040 <test_backtrace>: push   %ebp
13      {
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
=> 0xf0100040 <test_backtrace>: push   %ebp
test_backtrace (x=5) at kern/init.c:13
13      {
```

这是因为默认的gdb版本是8.1-0ubuntu3.1，需要修改成8.1-0ubuntu3才可以继续。这里就需要涉及到以下命令：

```
~$ sudo apt-get install gdb:8.1-0ubuntu3
~$ sudo apt-cache policy gdb
```

2. 新的gdb命令，第一条可以直接追踪到函数里面。第二条可以跳一行C语言代码，第三条可以打印从内存开始的固定长度的内容。

```
(gdb) b test_backtrace
(gdb) n
(gdb) x/64w $esp
```

解答

首先看看这一部分的C代码：

```
void test_backtrace(int x)
{
    cprintf("entering test_backtrace %d\n", x);
    if (x > 0)
        test_backtrace(x - 1);
    else
        mon_backtrace(0, 0, 0);
    cprintf("leaving test_backtrace %d\n", x);
}
```

这样其实就很简单了：每次进去都会调用一个`cprintf`告诉你“我进去了”，出来的时候也会和你说一声，接下来进入汇编。但是直接打断点的话会出现一个问题，解决方法在前置知识里面。

首先在`test_backtrace`打上断点，然后进入对应区域的时候不要急着下一步，因为在进入之前还有一些事情要做，看下面的代码就知道了，这段代码在`i386_init`里面。

```
f01000e8:   c7 04 24 05 00 00 00    movl   $0x5,(%esp)
f01000ef:   e8 4c ff ff ff          call   f0100040 <test_backtrace>
f01000f4:   83 c4 10                add    $0x10,%esp
```

这个时候来看看断点处的寄存器状态。为了节约篇幅，只列出涉及到的。

```
(gdb) i registers
ebx            0xf0111308       -267316472
esp            0xf010ffdc       0xf010ffdc
ebp            0xf010fff8       0xf010fff8
esi            0x10094  65684
eip            0xf0100040       0xf0100040 <test_backtrace>
```

然后看一下栈的`$esp`附近有啥东西，因为栈是从大地址往小地址走，所以不会存在问题。（一开始直接惯用64w的，结果发现太多了，就少拿了点。）后续都是这么操作，就不再多说了。

从下面的输出可以看出来，这里压入了两个整数，一个是主动压入的5，一个是自动压入的返回地址。

```
(gdb) x/64w $esp
0xf010ffdc:     0xf01000f4      0x00000005
```

进入`test_backtrace`的时候，首先是压栈和调用`cprintf`，如果进去就麻烦了，所以这里推荐使用gdb的`n`指令来一行C一行的跳。

容易理解的是第一个东西是调用的参数x，第二个是跳转出来的地址，附近的代码这里也贴出来。

```
f0100040:   55                      push   %ebp
f0100041:   89 e5                   mov    %esp,%ebp
f0100043:   56                      push   %esi
f0100044:   53                      push   %ebx
f0100045:   e8 72 01 00 00          call   f01001bc <__x86.get_pc_thunk.bx>
f010004a:   81 c3 be 12 01 00       add    $0x112be,%ebx
f0100050:   8b 75 08                mov    0x8(%ebp),%esi
    cprintf("entering test_backtrace %d\n", x);
f0100053:   83 ec 08                sub    $0x8,%esp
f0100056:   56                      push   %esi
f0100057:   8d 83 78 08 ff ff       lea    -0xf788(%ebx),%eax
f010005d:   50                      push   %eax
f010005e:   e8 ac 0a 00 00          call   f0100b0f <cprintf>
```

在第一句执行完了之后到了`sub $0x8, %esp`处，看看发生了什么。

```
ebx            0xf0111308       -267316472
esp            0xf010ffd0       0xf010ffd0
ebp            0xf010ffd8       0xf010ffd8
esi            0x5      5
eip            0xf0100053       0xf0100053 <test_backtrace+19>
0xf010ffd0:     0xf0111308      0x00010094      0xf010fff8      0xf01000f4
0xf010ffe0:     0x00000005
```

可以看到的是，相对于之前，新压入了三个变量，不难想到是三个push带来的。其中最妙的是在压入ebp之后立即把esp赋给ebp，这样在回传的时候就可以通过ebp自己构成链表循环下去。

```
ebx            0xf0111308       -267316472
esp            0xf010ffc0       0xf010ffc0
ebp            0xf010ffd8       0xf010ffd8
esi            0x5      5
eip            0xf0100063       0xf0100063 <test_backtrace+35>
0xf010ffc0:     0xf0101a80      0x00000005      0x00000000      0xf010004a
0xf010ffd0:     0xf0111308      0x00010094      0xf010fff8      0xf01000f4
0xf010ffe0:     0x00000005
```

按照代码可以知道，首先`$esp-=8`，然后又压入了两个东西，所以看到的是又变多了四个变量。这四个变量的来历中，最后一个`0xf010004a`可能是一个指针地址，由`$esp-=8`暴露出来，通过代码追踪可以发现是`call .get_pc_thunk`返回以后的下一步地址。第一个则是通过lea指令赋上的`$eax`压进去的结果。这些主要是给`cprintf`用的，用完了就是`$esp+=0x10`，消失。

之后可以预见的是发生了递归，通过`si`进去看看。在发现进去了以后立刻查看输出。在这之前，按`$esp+=0x10`，之后就是跳转，通过`$esp-=0x0c`暴露出来三个变量(4a~05)，接着又压进去了`$eax`，此时`$eax`已经为4。由于断点也会执行，所以说这里就会看到多一个东西压了进去，实际上，这个地址附近的代码也拿出来看看就都明白了。

```
ebx            0xf0111308       -267316472
esp            0xf010ffbc       0xf010ffbc
ebp            0xf010ffd8       0xf010ffd8
esi            0x5      5
eip            0xf0100040       0xf0100040 <test_backtrace>
0xf010ffbc:     0xf01000a1      0x00000004      0x00000005      0x00000000
0xf010ffcc:     0xf010004a      0xf0111308      0x00010094      0xf010fff8
0xf010ffdc:     0xf01000f4      0x00000005
```

附近的代码：

```
test_backtrace(x-1);
f0100095:   83 ec 0c                sub    $0xc,%esp
f0100098:   8d 46 ff                lea    -0x1(%esi),%eax
f010009b:   50                      push   %eax
f010009c:   e8 9f ff ff ff          call   f0100040 <test_backtrace>
f01000a1:   83 c4 10                add    $0x10,%esp
f01000a4:   eb d5                   jmp    f010007b <test_backtrace+0x3b>
```

之后基本所有的东西按照预期执行，就不再多阐述了。不过推荐在`0xf010095`看看输出会更有感觉一点。当然，这里有很多没什么用的内存。

```
0xf010ff70:     0xf0111308      0x00000003      0xf010ff98      0xf01000a1
0xf010ff80:     0x00000002      0x00000003      0xf010ffb8      0xf010004a
0xf010ff90:     0xf0111308      0x00000004      0xf010ffb8      0xf01000a1
0xf010ffa0:     0x00000003      0x00000004      0x00000000      0xf010004a
0xf010ffb0:     0xf0111308      0x00000005      0xf010ffd8      0xf01000a1
0xf010ffc0:     0x00000004      0x00000005      0x00000000      0xf010004a
0xf010ffd0:     0xf0111308      0x00010094      0xf010fff8      0xf01000f4
0xf010ffe0:     0x00000005
```

我们来看看不停的跳转之后如何收回这些空间的。经过不断的分析，终于到了这个需要回收的地方，`mon_backtrace`，我们先看看内存会扩展成什么样子。

```
0xf010ff20:     0x00000000      0x00000000      0x00000000      0xf010004a
0xf010ff30:     0xf0111308      0x00000001      0xf010ff58      0xf01000a1
0xf010ff40:     0x00000000      0x00000001      0xf010ff78      0xf010004a
0xf010ff50:     0xf0111308      0x00000002      0xf010ff78      0xf01000a1
0xf010ff60:     0x00000001      0x00000002      0xf010ff98      0xf010004a
0xf010ff70:     0xf0111308      0x00000003      0xf010ff98      0xf01000a1
0xf010ff80:     0x00000002      0x00000003      0xf010ffb8      0xf010004a
0xf010ff90:     0xf0111308      0x00000004      0xf010ffb8      0xf01000a1
0xf010ffa0:     0x00000003      0x00000004      0x00000000      0xf010004a
0xf010ffb0:     0xf0111308      0x00000005      0xf010ffd8      0xf01000a1
0xf010ffc0:     0x00000004      0x00000005      0x00000000      0xf010004a
0xf010ffd0:     0xf0111308      0x00010094      0xf010fff8      0xf01000f4
0xf010ffe0:     0x00000005
```

到这里的下一步就是调用`mon_backtrace`，到这里可以猜测asm代码的真正用意，实际上有很多是为了填充，填充成四字节方便清空，同时也可以很好防止内存攻击。这里就有很多人想问了，为什么会有两个操作来统一`$esp`的大小？第一个岂不是白干了？我们先放下疑问继续看。

通过这样的操作，成功把最后一次调用的栈给退了回去，然后就是三个`pop`，顺序和`push`是相反的。继续`n`一次直接跳过，可以发现内存小了很多。

```
0xf010ff50:     0xf0111308      0x00000002      0xf010ff78      0xf01000a1
0xf010ff60:     0x00000001      0x00000002      0xf010ff98      0xf010004a
0xf010ff70:     0xf0111308      0x00000003      0xf010ff98      0xf01000a1
0xf010ff80:     0x00000002      0x00000003      0xf010ffb8      0xf010004a
0xf010ff90:     0xf0111308      0x00000004      0xf010ffb8      0xf01000a1
0xf010ffa0:     0x00000003      0x00000004      0x00000000      0xf010004a
0xf010ffb0:     0xf0111308      0x00000005      0xf010ffd8      0xf01000a1
0xf010ffc0:     0x00000004      0x00000005      0x00000000      0xf010004a
0xf010ffd0:     0xf0111308      0x00010094      0xf010fff8      0xf01000f4
0xf010ffe0:     0x00000005
```

少了两行，这一部分到底发生了什么，我们`si`看看发现了这个东西。

```
(gdb) si
=> 0xf01000a1 <test_backtrace+97>:      add    $0x10,%esp
0xf01000a1 in test_backtrace (x=2) at kern/init.c:16
16                      test_backtrace(x-1);
```

这下真相就大白了。回去的整个调用过程通过gdb可以看得一清二楚。在mon结束之后，继续执行二进制码到call位置调用`cprintf`，接着运行到`ret`，跳转回上一个调用的地址。但是发现这个地址已经被单独在函数外了，所以通过一个`jmp`跳回函数内，并且`$esp+=0x10`来清空部分栈。剩下的就是`cprintf`的事情了，它又拿走了16bytes，又被读了回来。

最后可能有人会问，每次操作都是16bytes，那么

```
f010008b:   83 c4 10                add    $0x10,%esp
f010008e:   8d 65 f8                lea    -0x8(%ebp),%esp
f0100091:   5b                      pop    %ebx
f0100092:   5e                      pop    %esi
f0100093:   5d                      pop    %ebp
f0100094:   c3                      ret
```

这一段代码是怎么回事?

实际上回答很简单，因为`ret`自动pop了一个东西。至于原因，可以gdb调试看看，我把附近的东西都拿了出来，相信你们可以发现规律。那个x/64w就不用吐槽了，每次算这么多我也很烦，所以干脆一次性弄多点。

```
(gdb) si
=> 0xf0100093 <test_backtrace+83>:      pop    %ebp
0xf0100093      20      }
(gdb) x/64w $esp
0xf010ff58:     0xf010ff78      0xf01000a1      0x00000001      0x00000002
0xf010ff68:     0xf010ff98      0xf010004a      0xf0111308      0x00000003
0xf010ff78:     0xf010ff98      0xf01000a1      0x00000002      0x00000003
0xf010ff88:     0xf010ffb8      0xf010004a      0xf0111308      0x00000004
0xf010ff98:     0xf010ffb8      0xf01000a1      0x00000003      0x00000004
0xf010ffa8:     0x00000000      0xf010004a      0xf0111308      0x00000005
0xf010ffb8:     0xf010ffd8      0xf01000a1      0x00000004      0x00000005
0xf010ffc8:     0x00000000      0xf010004a      0xf0111308      0x00010094
0xf010ffd8:     0xf010fff8      0xf01000f4      0x00000005      0x00001aac
(gdb) si
=> 0xf0100094 <test_backtrace+84>:      ret
0xf0100094      20      }
(gdb) x/64w $esp
```

```
0xf010ff5c:     0xf01000a1      0x00000001      0x00000002      0xf010ff98
0xf010ff6c:     0xf010004a      0xf0111308      0x00000003      0xf010ff98
0xf010ff7c:     0xf01000a1      0x00000002      0x00000003      0xf010ffb8
0xf010ff8c:     0xf010004a      0xf0111308      0x00000004      0xf010ffb8
0xf010ff9c:     0xf01000a1      0x00000003      0x00000004      0x00000000
0xf010ffac:     0xf010004a      0xf0111308      0x00000005      0xf010ffd8
0xf010ffbc:     0xf01000a1      0x00000004      0x00000005      0x00000000
0xf010ffcc:     0xf010004a      0xf0111308      0x00010094      0xf010fff8
0xf010ffdc:     0xf01000f4      0x00000005      0x00001aac
(gdb) si
=> 0xf01000a1 <test_backtrace+97>:      add    $0x10,%esp
0xf01000a1 in test_backtrace (x=2) at kern/init.c:16
16                      test_backtrace(x-1);
(gdb) x/64w $esp
0xf010ff60:     0x00000001      0x00000002      0xf010ff98      0xf010004a
0xf010ff70:     0xf0111308      0x00000003      0xf010ff98      0xf01000a1
0xf010ff80:     0x00000002      0x00000003      0xf010ffb8      0xf010004a
0xf010ff90:     0xf0111308      0x00000004      0xf010ffb8      0xf01000a1
0xf010ffa0:     0x00000003      0x00000004      0x00000000      0xf010004a
0xf010ffb0:     0xf0111308      0x00000005      0xf010ffd8      0xf01000a1
0xf010ffc0:     0x00000004      0x00000005      0x00000000      0xf010004a
0xf010ffd0:     0xf0111308      0x00010094      0xf010fff8      0xf01000f4
0xf010ffe0:     0x00000005
```

Lab1的exercise11

问题

实现如上所述的回溯功能。请使用与示例中相同的格式，否则打分脚本将会出错。当你认为你的工作正确的时候，运行`make grade`来看看它的输出是否符合我们的打分脚本的期待，如果没有，修正发现的错误。在你成功提交实验 1 的作业后，欢迎你以任何你喜欢的方式更改回溯功能的输出格式。

如果你使用`read_ebp()`，请注意，GCC 可能会生成优化后的代码，导致在`mon_backtrace()`的函数前导代码之前调用`read_ebp()`，从而导致堆栈跟踪不完整（最近的函数调用的堆栈帧丢失）。我们可以尝试禁用优化以避免此重新排序的优化模式。如果你遇到了这个问题，不需要解决它，只要能够解释它即可。你可能需要检查`mon_backtrace()`的汇编代码，并确保在函数前导代码之后调用的`read_ebp()`。

前置知识

1. 一些寄存器的内容

   寄存器eip负责告诉CPU在调用出来了以后跳转到哪里。

   寄存器ebp告诉CPU栈底在哪里。

   寄存器esp告诉CPU这个函数在进入之后栈顶在哪里。

2. 可以通过`read_ebp()`发送asm来读取ebp寄存器的内容。

解答

在这里首先要明白的是，系统在运行的时候如果发生了调用，会用到三种寄存器。接下来？直接开始写就行了。根据前面的可以知道，读取`ebp = read_ebp()`，然后把`ebp[1]~ebp[6]`拿出来就行了，这里`eip`刚好就是返回指令指针。

```
int mon_backtrace(int argc, char **argv, struct Trapframe *tf)
```

```
{
    // Your code here.
    uint32_t *ebp;
    ebp = (uint32_t *) read_ebp();
    cprintf("Stack backtrace:\r\n");
    while(ebp)
    {
        cprintf("  ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\r\n",
            ebp, ebp[1], ebp[2], ebp[3], ebp[4], ebp[5], ebp[6]);
        ebp = (uint32_t *)*ebp;
    }
    return 0;
}
```

最后运行一下看看结果：

```
ivy233@DESKTOP-LVJJABH:~/lab$ make qemu-nox
***
*** Use Ctrl-a x to exit qemu
***
/usr/local/qemu/bin/qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -serial mon:stdio -gdb tcp::26000 -D qemu.log
6828 decimal is 15254 octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
Stack backtrace:
  ebp f010ff18  eip f0100078  args 00000000 00000000 00000000 f010004a f0111308
  ebp f010ff38  eip f01000a1  args 00000000 00000001 f010ff78 f010004a f0111308
  ebp f010ff58  eip f01000a1  args 00000001 00000002 f010ff98 f010004a f0111308
  ebp f010ff78  eip f01000a1  args 00000002 00000003 f010ffb8 f010004a f0111308
  ebp f010ff98  eip f01000a1  args 00000003 00000004 00000000 f010004a f0111308
  ebp f010ffb8  eip f01000a1  args 00000004 00000005 00000000 f010004a f0111308
  ebp f010ffd8  eip f01000f4  args 00000005 00001aac 00000640 00000000 00000000
  ebp f010fff8  eip f010003e  args 00000003 00001003 00002003 00003003 00004003
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

Lab1的exercise12

问题

题目太长，这里简略描述。这里主要涉及到三个问题。

1. 一些寄存器的内容在 `debuginfo_eip` 中 `_STAB` 来自哪里?
2. 通过插入对 `stab_binsearch` 的调用来查找地址的行号，补全 `debuginfo_eip` 的实现。
3. 向内核监视器添加一个 `backtrace` 命令，并扩展你的 `mon_backtrace` 的实现，调用 `debuginfo_eip` 并为每个堆栈框架打印一行:

解答

在回答这些问题之前，首先把代码填充完整，然后再来看看这个STAB是个什么东西。

首先把 `kern/kdebug.c` 中第181行附近的添加代码加上这些

```
stab_binsearch(stabs, &lline, &rline, N_SLINE,
               addr - info->eip_fn_addr);
if (lline <= rline) {
    info->eip_line = stabs[lline].n_desc;
} else {
    return -1;
}
```

然后把 `mon_backtrace` 有样学样地加上(`kern/monitor.c`):

```
static struct Command commands[] = {
    {"help", "Display this list of commands", mon_help},
    {"kerninfo", "Display information about the kernel", mon_kerninfo},
    {"backtrace", "Display backtrace info", mon_backtrace}};
```

最后修改 `mon_backtrace` ，原来是这个代码:

```
int mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    uint32_t *ebp;
    ebp = (uint32_t *) read_ebp();
    cprintf("Stack backtrace:\r\n");
    while(ebp)
    {
        cprintf("  ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\r\n",
ebp, ebp[1], ebp[2], ebp[3], ebp[4], ebp[5], ebp[6]);
        ebp = (uint32_t *)*ebp;
    }
    return 0;
}
```

修改为以下代码:

```
int mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    uint32_t *ebp;
```

```
    struct Eipdebuginfo info;
    int result;
    ebp = (uint32_t *) read_ebp();
    cprintf("Stack backtrace:\r\n");
    while(ebp)
    {
cprintf("  ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\r\n",
     ebp, ebp[1], ebp[2], ebp[3], ebp[4], ebp[5], ebp[6]);
        memset(&info, 0, sizeof(struct Eipdebuginfo));
        result = debuginfo_eip(ebp[1], &info);
        if (0 != result) {
            cprintf("failed to get debuginfo for eip %x.\r\n", ebp[1]);
        } else {
            cprintf("\t%s:%d: %.*s+%u\r\n", info.eip_file,
                info.eip_line, info.eip_fn_namelen, info.eip_fn_name,
                ebp[1] - info.eip_fn_addr);
        }
        ebp = (uint32_t *)*ebp;
    }
    return 0;
}

```

最后在terminal运行一下`make grade`检查结果。

```
make[1]: Leaving directory '/home/ivy233/lab'
running JOS: (1.9s)
  printf: OK
  backtrace count: OK
  backtrace arguments: OK
  backtrace symbols: OK
  backtrace lines: OK
Score: 50/50
```

如果想直接获得结果看到这里就可以结束了。

但是作为一个做实验的，怎么可能到这里为止呢？接下来我们按照实验要求给的提示，从顶往下调用看看到底发生了什么。做这个Exercise一定要注意不要按照实验参考给的顺序来完成实验，最好反过来处理。

我们从入口开始，首先是`mon_backtrace`。下面贴的是之前实现的代码。可以对比一下差异。从实验要求可以看出，要求新增这个是在哪里实行的，哪个文件，哪一行，内存中的哪个位置。

这个时候回头来看差异在哪里。新增了一个`Eipdebuginfo`结构，每次清空，在里面查找什么东西，然后如果不存在要报错，反过来存在的话就要输出什么东西，从名称来看，正好是我们想要的：文件，行号，函数名，相对地址。也就是说`debuginfo_eip`这个函数用地址查询了刚才说的信息并且把它塞进info这个结构体。

接下来追下去看看这个地址怎么查出来的。代码太长就不放出来了，这里只给出关键的部分

```
lfile = 0;
rfile = (stab_end - stabs) - 1;
stab_binsearch(stabs, &lfile, &rfile, N_SO, addr);
if (lfile == 0)
    return -1;

lfun = lfile;
```

```
    rfun = rfile;
    stab_binsearch(stabs, &lfun, &rfun, N_FUN, addr);

    if (lfun <= rfun) {
        if (stabs[lfun].n_strx < stabstr_end - stabstr)
            info->eip_fn_name = stabstr + stabs[lfun].n_strx;
        info->eip_fn_addr = stabs[lfun].n_value;
        addr -= info->eip_fn_addr;
        lline = lfun;
        rline = rfun;
    } else {
        info->eip_fn_addr = addr;
        lline = lfile;
        rline = rfile;
    }
    info->eip_fn_namelen = strfind(info->eip_fn_name, ':') -
        info->eip_fn_name;
    stab_binsearch(stabs, &lline, &rline, N_SLINE, addr -
                   info->eip_fn_addr);
    if (lline <= rline) {
        info->eip_line = stabs[lline].n_desc;
    } else {
        return -1;
    }
```

这一段代码最为关键，完成了从文件->函数->行号的搜索，至于具体的搜索方式则大致为对地址二分搜索，用TYPE为搜索关键词，很容易就能想到地址是从低到高排列的。除此以外，这个STAB有结构的解析，就是`<inc/stab.h>`里面说的（见下）。除此之外还有很多的常量定义，都是和TYPE有关的。

```
// Entries in the STABS table are formatted as follows.
struct Stab {
    uint32_t n_strx;     // index into string table of name
    uint8_t n_type;      // type of symbol
    uint8_t n_other;     // misc info (usually empty)
    uint16_t n_desc;     // description field
    uintptr_t n_value;   // value of symbol
};
```

说了这么多，STAB存了什么东西?

STAB，全程signed table，就是符号表，和编译原理一样，需要里面放进去一些东西才能辅助编译。这一点和gdb的代码定位如出一辙。

存放的内容通过这条命令可以打出来，不过内容很长，这里就放出一点内容，-1到20。

```
ivy233@DESKTOP-LVJJABH:~/lab$ objdump -G obj/kern/kernel
-1      HdrSym 0      1302   0000198b 1
0       SO     0      0      f0100000 1      {standard input}
1       SOL    0      0      f010000c 18     kern/entry.S
2       SLINE  0      44     f010000c 0
3       SLINE  0      57     f0100015 0
4       SLINE  0      58     f010001a 0
5       SLINE  0      60     f010001d 0
6       SLINE  0      61     f0100020 0
7       SLINE  0      62     f0100025 0
8       SLINE  0      67     f0100028 0
9       SLINE  0      68     f010002d 0
```

```
10     SLINE  0      74     f010002f 0
11     SLINE  0      77     f0100034 0
12     SLINE  0      80     f0100039 0
13     SLINE  0      83     f010003e 0
14     SO     0      2      f0100040 31     kern/entrypgdir.c
15     OPT    0      0      00000000 49     gcc2_compiled.
16     LSYM   0      0      00000000 64
int:t(0,1)=r(0,1);-2147483648;2147483647;
17     LSYM   0      0      00000000 106    char:t(0,2)=r(0,2);0;127;
18     LSYM   0      0      00000000 132    long
int:t(0,3)=r(0,3);-2147483648;2147483647;
19     LSYM   0      0      00000000 179    unsigned
int:t(0,4)=r(0,4);0;4294967295;
20     LSYM   0      0      00000000 220    long unsigned
int:t(0,5)=r(0,5);0;4294967295;
```

仔细查看去中的内容，可以发现SOL/SO就是对应的文件，接下来的就是文件内的地址或者其他什么东西。而二分搜索的内容就是从这里面找的。

然后就可以看看实际运行的时候发生了什么，在这之前，需要看看到底存放到哪个地址了。

```
ivy233@DESKTOP-LVJJABH:~/lab$ objdump -h obj/kern/kernel
obj/kern/kernel:     file format elf32-i386
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00001b69  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       0000077c  f0101b80  00101b80  00002b80  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         00003d15  f01022fc  001022fc  000032fc  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      0000198c  f0106011  00106011  00007011  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         00009300  f0108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .got          00000008  f0111300  00111300  00012300  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  6 .got.plt      0000000c  f0111308  00111308  00012308  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  7 .data.rel.local 00001000  f0112000  00112000  00013000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  8 .data.rel.ro.local 00000060  f0113000  00113000  00014000  2**5
                  CONTENTS, ALLOC, LOAD, DATA
  9 .bss          00000648  f0113060  00113060  00014060  2**5
                  CONTENTS, ALLOC, LOAD, DATA
 10 .comment      0000002b  00000000  00000000  000146a8  2**0
                  CONTENTS, READONLY
```

可以看到的是存放在`0xf0106011`这个地方，我们就进去看看。之后通过在不同的地方测试，可以知道的是：

```
(gdb) b *0x100025
Note: breakpoint 2 also set at pc 0x100025.
Breakpoint 3 at 0x100025
(gdb) x/8s 0xf0106011
0xf0106011:     ""
```

```
0xf0106012:     ""
0xf0106013:     ""
0xf0106014:     ""
0xf0106015:     ""
0xf0106016:     ""
0xf0106017:     ""
0xf0106018:     ""
(gdb) b test_backtrace
Breakpoint 4 at 0xf0100040: file kern/init.c, line 13.
(gdb) c
Continuing.
=> 0xf0100040 <test_backtrace>: push   %ebp

Breakpoint 4, test_backtrace (x=5) at kern/init.c:13
13      {
(gdb) x/8s 0xf0106011
0xf0106011:     ""
0xf0106012:     "{standard input}"
0xf0106023:     "kern/entry.S"
0xf0106030:     "kern/entrypgdir.c"
0xf0106042:     "gcc2_compiled."
0xf0106051:     "int:t(0,1)=r(0,1);-2147483648;2147483647;"
0xf010607b:     "char:t(0,2)=r(0,2);0;127;"
0xf0106095:     "long int:t(0,3)=r(0,3);-2147483648;2147483647;"
```

这样就能明白这个是干嘛了的。

至此，lab1结束。用git提交以下就可以完成了。

```
~/lab$ git commit -am "xxx"
~/lab$ git checkout -b lab2 origin/lab2
~/lab$ git merge lab1
```