**哈尔滨工业大学计算机科学与技术学院**

**2017年秋季学期《操作系统》**

**实验报告**

|  |  |  |
| --- | --- | --- |
| **姓名** | **学号** | **联系方式** |
| **卢霄晗** | **1150320206** | **739651656@qq.com** |

# 一、xv6实验系统简述

（正文采用宋体四号字，行距1.25倍，段前0行，段后0行。）

# 二、xv6环境配置（附过程截图）

# 三、实验过程（lab1~lab6）

## Lab 1: Booting a PC

### 实验目的

### 实验内容（附过程截图）

* 1. 前期准备

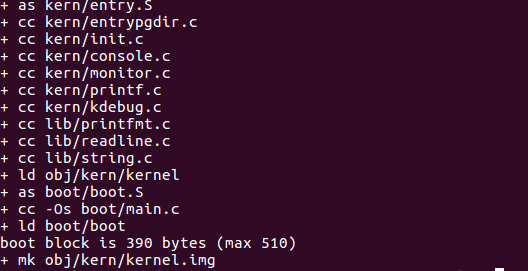
系统：Ubuntu 14.04 64位，在vmware虚拟机中运行

已安装qemu模拟器

导入JOS源码

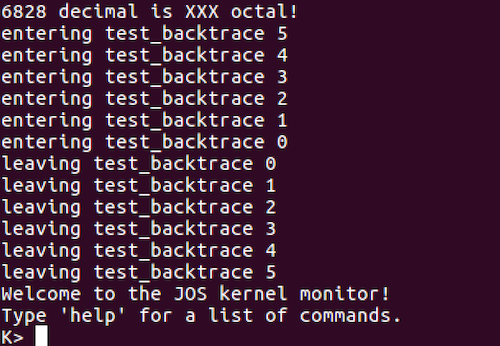
# git clone https://pdos.csail.mit.edu/6.828/2017/jos.git lab

# make install



生成kernel.img镜像

# make qemu 用qemu去运行镜像



环境搭建完成

* 1. PC Bootstrap

**Exercise 1**：熟悉汇编语言，参考reference即可，不做详细描述

**Exercise 2**：

在一个终端中输入 # make qemu-gdb ， 另一个终端输入 # make gdb 。开始调试程序。

[f000:fff0] 0xffff0: ljmp $0xf000,$0xe05b

是GDB反汇编出的第一条执行指令，这条指令表面了：

IBM PC 执行的起始物理地址为 0x000ffff0

PC 的偏移方式为 CS = 0xf000，IP = 0xfff0

第一条指令执行的是 jmp指令，跳转到段地址 CS = 0xf000，IP = 0xe05b

QEMU模拟了8088处理器的启动，当启动电源，BIOS最先控制机器，这时还没有其他程序执行，之后处理器进入实模式也就是设置 CS 为 0xf000，IP 为 0xfff0。在启动电源也就是实模式时，地址转译根据这个公式工作：物理地址 = 16 \* 段地址 + 偏移量。所以 PC 中 CS 为 0xf000 IP 为 0xfff0 的物理地址为：

16 \* 0xf000 + 0xfff0 # 十六进制中乘16很容易

= 0xf0000 + 0xfff0 # 仅仅添加一个0.

= 0xffff0

0xffff0 在 BIOS (0x100000) 的结束地址之前。

当BIOS启动，它设置了一个中断描述符表并初始化多个设备比如VGA显示器。在初始化PCI总线和所有重要的设备之后，它寻找可引导的设备，之后读取 boot loader 并转移控制。

* 1. The Boot Loader

功能：

1. boot loader 切换处理器从实模式到保护模式，只有这样才能访问大于1MB的物理地址空间。
2. boot loader 从硬盘中读取内核。

**Exercise 3**：通过 b \*0x7c00 设置断点，接着 c 运行到断点处，使用 x/i 来查看当前的指令。

3.1. At what point does the processor start executing 32-bit code?What exactly causes the switch from 16- to 32-bit mode?

[ 0:7c2d] => 0x7c2d: ljmp $0x8,$0x7c32 这条指令之后，也就是 boot.S 中的 ljmp $PROT\_MODE\_CSEG, $protcseg ，地址符号就变成 0x7c32 了。

3.2. What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?

boot loader 最后一步是加载kernel，所以在 boot/main.c 中可以找到 ((void (\*)(void)) (ELFHDR->e\_entry))(); 这行代码，上面的注释 call the entry point from the ELF header 表明这是准备读取ELF头。  
通过 objdump -x obj/kern/kernel 可以查看kernel的信息，其中开头就有 start address 0x0010000c，通过 b \*0x10000c然后在 c 能得到执行的指令是 movw $0x1234,0x472，当然在 kern/entry.S 中也能找到这个指令。

3.3. Where is the first instruction of the kernel?

同上一条的问题，存在kern/entry.S中。

3.4. How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

关于查看kernel信息的，通过 objdump -h obj/kern/kernel 可以得出相关信息。

接下来就可以来完成 Exercise 3。设置一个断点在地址0x7c00处，这是boot sector被加载的位置。然后让程序继续运行直到这个断点。跟踪/boot/boot.S文件的每一条指令，同时使用boot.S文件和系统为你反汇编出来的文件obj/boot/boot.asm。你也可以使用GDB的x/i指令来获取去任意一个机器指令的反汇编指令，把源文件boot.S文件和boot.asm文件以及在GDB反汇编出来的指令进行比较。

追踪到bootmain函数中，而且还要具体追踪到readsect()子函数里面。找出和readsect()c语言程序的每一条语句所对应的汇编指令，回到bootmain()，然后找出把内核文件从磁盘读取到内存的那个for循环所对应的汇编语句。找出当循环结束后会执行哪条语句，在那里设置断点，继续运行到断点，然后运行完所有的剩下的语句。

首先查看boot.S文件，在开头可以看到

start:

.code16 # 16位汇编模式

cli # 关中断

cld # String operations increment

cld 是串操作指令，用来操作方向标志位DF，使DF=0。

# 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

将DS、ES、SS寄存器清零。

# 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

开启A20，关于为什么要开A20,可以看[知乎:OS boot 的时候为什么要 enable A20？](https://www.zhihu.com/question/29375534)。

inb $0x64,%al 把0x64端口(8042键盘控制器)的状态写入al中（inb代表IO端口读）, 之后 testb $0x2,%al 判断al的第二位是否为0，不为0就循环执行seta20.1。这里第二位代表输入缓冲区是否满了。接着0xd1放入0x64端口。最后将0xdf放入0x60端口，代表开启A20地址线了。

# 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

切换到保护模式后，加载GDT(Global Descriptor Table)，接着修改了cr0寄存器的值，$CR0\_PE\_ON值为0x1，代表启动保护模式的flag标志。

# Jump to next instruction, but in 32-bit code segment.

# Switches processor into 32-bit mode.

ljmp $PROT\_MODE\_CSEG, $protcseg

也就是 0x7c2d: ljmp $0x8,$0x7c32 跳转到了32位代码段。

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

修改了这些寄存器的值。

# Set up the stack pointer and call into C.

movl $start, %esp

call bootmain

设置栈指针，接着开始调用bootmain函数。

7d15: 55 push %ebp

7d16: 89 e5 mov %esp,%ebp

7d18: 56 push %esi

7d19: 53 push %ebx

首先做进入函数的准备工作

// read 1st page off disk

readseg((uint32\_t) ELFHDR, SECTSIZE\*8, 0);

7d1a: 6a 00 push $0x0

7d1c: 68 00 10 00 00 push $0x1000

7d21: 68 00 00 01 00 push $0x10000

7d26: e8 b1 ff ff ff call 7cdc <readseg>

接着调用readseg函数，这个函数有3个参数，第一个是物理地址，第二个是页的大小，第三个是偏移量。

0x7ceb: shr $0x9,%edi 执行了 offset = (offset / SECTSIZE) + 1; 这条代码前面的除法部分，得出扇区号。

0x7cee: add %ebx,%esi 执行了 end\_pa = pa + count; 计算出这个扇区结束的物理地址。

0x7cf0: inc %edi 执行了 offset = (offset / SECTSIZE) + 1; 中的加1。

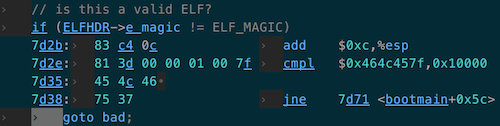
0x7cf1: and $0xfffffe00,%ebx 执行了 pa &= ~(SECTSIZE - 1);。

0x7cf7: cmp %esi,%ebx

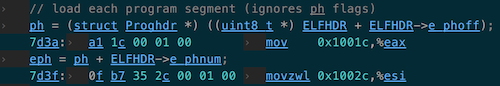
0x7cf9: jae 0x7d0d

执行 while (pa < end\_pa) 这个循环判断语句。

之后几条汇编是为readsect函数做准备，这个函数是读取扇区内容的。



判断 ELFHDR->e\_magic != ELF\_MAGIC 这个条件。



加载程序段由这几个部分汇编构成

7d3a: a1 1c 00 01 00 mov 0x1001c,%eax

7d3f: 0f b7 35 2c 00 01 00 movzwl 0x1002c,%esi

7d46: 8d 98 00 00 01 00 lea 0x10000(%eax),%ebx

7d4c: c1 e6 05 shl $0x5,%esi

7d4f: 01 de add %ebx,%esi

之后循环调用readseg函数，将Program Header Table中表项读入内存。

最后一步就是

((void (\*)(void)) (ELFHDR->e\_entry))();

以上为Exercise 3大致内容。

**加载内核**

**Exercise 4**

阅读 K&R 的 The C Programming Language， 理解 [pointers.c](https://pdos.csail.mit.edu/6.828/2016/labs/lab1/pointers.c).

int a[4];

int \*b = malloc(16);

int \*c;

int i;

printf("1: a = %p, b = %p, c = %p\n", a, b, c);

结果是 1: a = 0x7fff5fbff810, b = 0x100600000, c = 0x100000000

a是数组，所以输出的0x7fff5fbff810是数组a的首地址。b是指针，指向malloc分配的空间的起始地址0x100600000。c是未定义的指针。

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 = a; 让指针c指向a指向的地址。先是for循环，给数组a赋值为100、101、102、103。因为c现在和a指向同一区域，所以c[0]修改的是数组a的第一个元素，所以结果是 2: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103.

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]);

三种修改数组中值的方法，其中第三种没怎么见过，有点像汇编里地址偏移的方式，3[c]就是c的地址偏移到第三个元素的地址，所以输出为 3: a[0] = 200, a[1] = 300, a[2] = 301, a[3] = 302.

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]);

移动指针指向了数组的第二个元素，所以输出为 4: a[0] = 200, a[1] = 400, a[2] = 301, a[3] = 302.

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]);

c = (int \*) ((char \*) c + 1); 这一行代码先将c强制转型为char指针，接着指针加1，再强转回int指针。一步一步来看，首先可以确定在执行这行代码之前，c的地址为 0x7fff5fbff814(因为系统是64位的，其实地址为 0x00007fff5fbff814，只不过前面的0省略了)，所以强转为char指针后加1得到的地址为 0x7fff5fbff815，因为char只占一个字节。因为c原来值为400，也就是十六进制的0x190.在c强转之前，打印 (char ) c 的地址为 0x7fff5fbff814，查看 (char \*) c 的值为 ffffff90。这里要说明一下，为什么90前面都是ffffff，这不是fff团对单身狗的报复，而是

printf("%x\n", \*(char \*)c);

中，%x 将值强转为unsigned int，unsigned int在我的系统中占4个字节。所以打印出来有8位。用 %c 可以得到真正这个地方的值，只不过是八进制的。当然查看这个值最方便的还是声明个变量，然后在Xcode打断点进行查看。声明一个变量 char \*p= (char \*)c;, 使用 p++ 进行单步调试，可以查看 \*p 的值。

0x7fff5fbff814 0x90

0x7fff5fbff815 0x01

0x7fff5fbff816 0x0

0x7fff5fbff817 0x0

0x7fff5fbff818 2d

0x7fff5fbff819 1

0x7fff5fbff81a 0

0x7fff5fbff81b 0

500的十六进制是0x1F4, 因为char指针加1使指针指向 0x7fff5fbff815，所以修改其值为0xF4，0x7fff5fbff816为0x1，0x7fff5fbff818为0x0，所以a[1]值变为0x1F490=128144，a[2]为0x100=256。所以输出 a[0] = 200, a[1] = 128144, a[2] = 256, a[3] = 302。

b = (int \*) a + 1;

c = (int \*) ((char \*) a + 1);

printf("6: a = %p, b = %p, c = %p\n", a, b, c);

b是int指针，所以a + 1即a的地址加上sizeof(int \*)也就是4，所以指向a[1]。而c参考上一段的介绍，可知道是在char指针情况下进行加1，所以地址偏移1。所以输出 6: a = 0x7fff5fbff810, b = 0x7fff5fbff814, c = 0x7fff5fbff811。

接着就可以来学习内核。为了理解 boot/main.c， 需要了解ELF二进制文件。编译并链接比如JOS内核这样的C程序，编译器会将源文件(.c)转为包含汇编指令的目标文件(.o)。接着链接器把所有的目标文件组合成一个单独的二进制镜像（binary image），比如 obj/kern/kernel，这种文件就是ELF(是可执行可链接形式的缩写)。

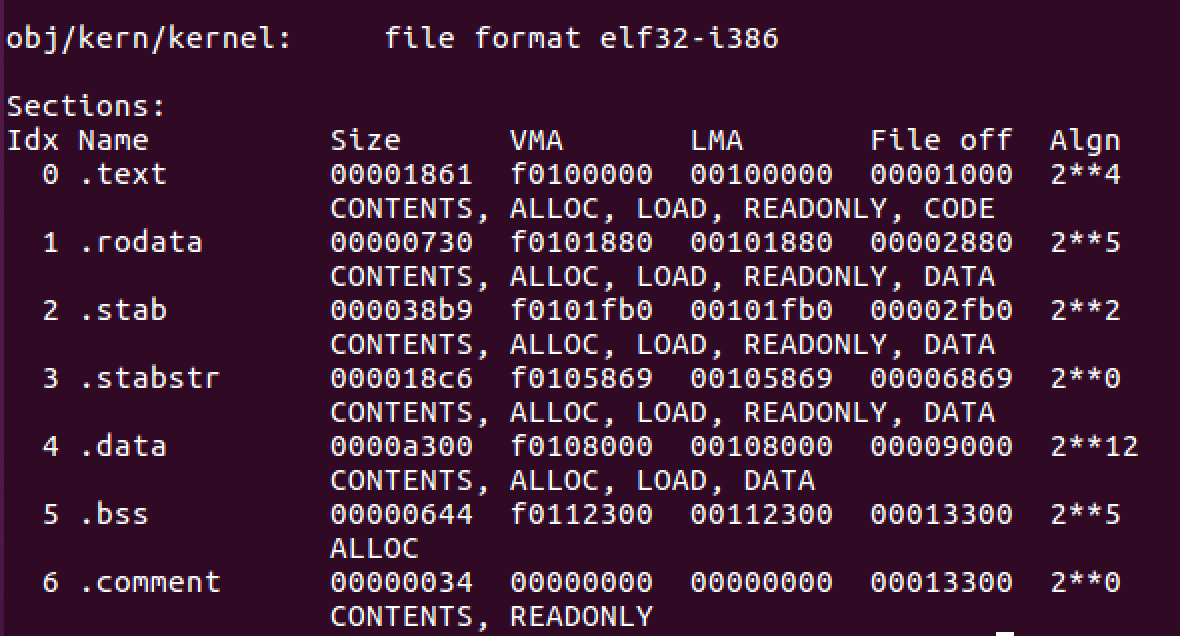
当前只需要知道，可执行的ELF文件由带有加载信息的头，多个程序段表组成。每个程序段表是一个连续代码块或者数据，它们要被加载到内存具体地址中。boot loader 不修改源码和数据，直接加载到内存中并运行。

ELF开头是固定长度的 ELF头，之后是一个可变长度的程序头，它列出了需要加载的程序段。ELF头的定义在 inc/elf.h 中。主要学习以下3个程序段：

* .text: 程序执行指令
* .rodata:只读数据，比如ASCII字符串
* .data: 存放程序初始化的数据段，比如有初始值的全局变量。

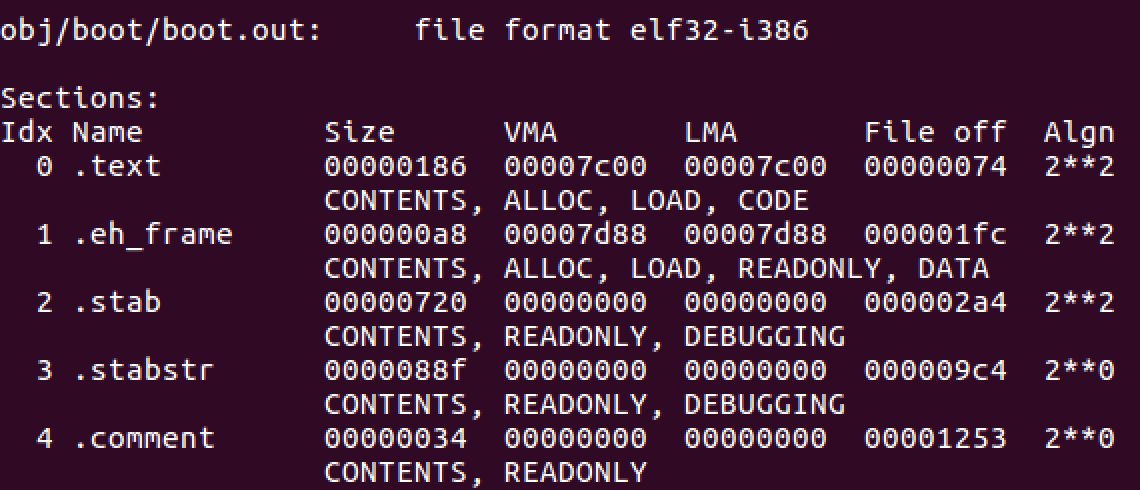
当链接器计算程序内存布局时，会在内存里紧挨着.data段的.bss段中保留空间给未初始化的全局变量。C规定未初始化的全局变量为0。因此没必要在ELF的.bss段储存内容，链接器只储存了.bss段的地址和大小。

使用 objdump -h obj/kern/kernel 可以查看ELF头的相关信息。



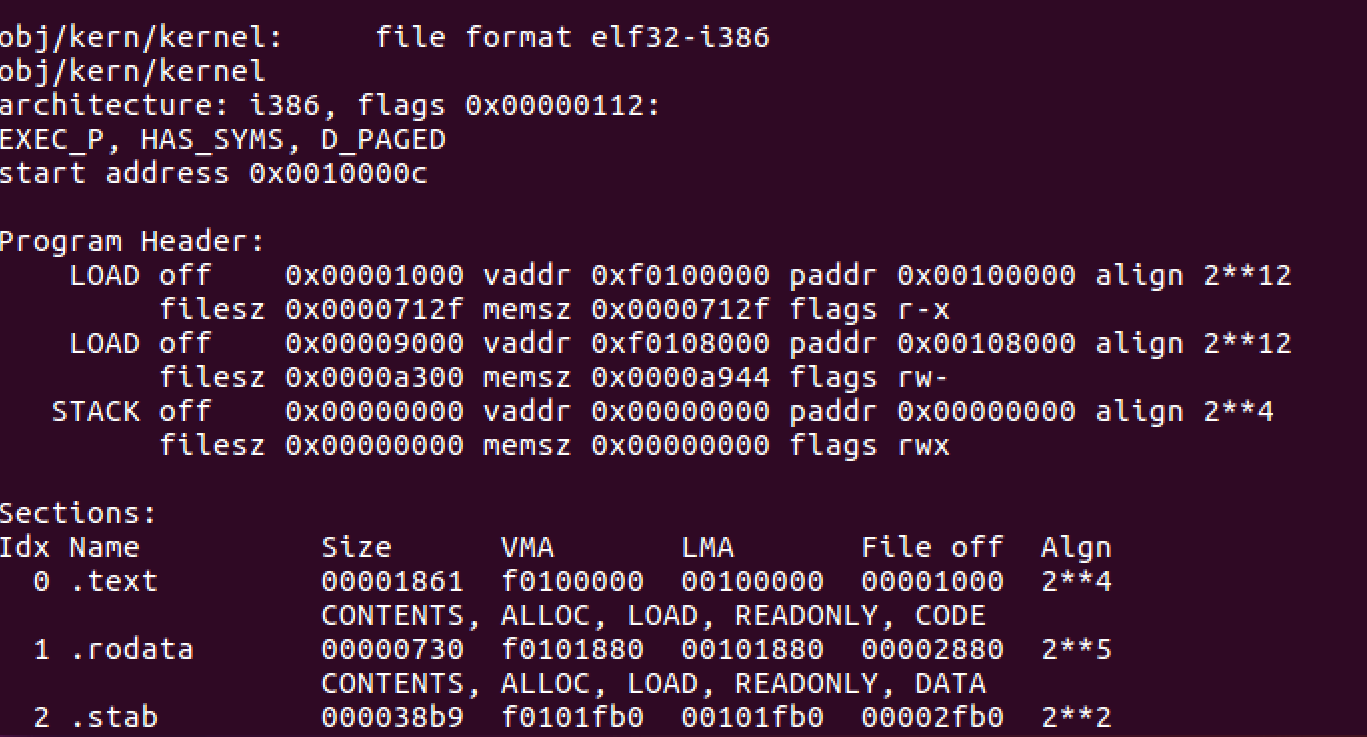
重点关注 .text段 的VMA(链接地址)和LMA(加载地址)，段的加载地址即加载进内存的地址。段的链接地址就是这个段预计在内存中执行的地址。链接程序有多种编码链接地址的方法。通常链接和加载的地址是一致的。查看boot loader的 .text段

objdump -h obj/boot/boot.out



boot loader使用 ELF程序头(Program Headers) 确定如何加载段。程序头指明ELF中哪部分加载进内存和其所在的地址。使用一下命令查看

objdump -x obj/kern/kernel



其中Program Headers下面列出的程序头中，开头的LOAD代表已经加载到内存中了，另外显示出了虚拟地址(vaddr)，物理地址(paddr)以及存放区域的大小(memsz和filesz)。

回到 boot/main.c， ph->p\_pa是每个程序头包含的段目的物理地址。

BIOS把引导扇区加载到内存地址0x7c00，这也就是引导扇区的加载地址和链接地址。在 boot/Makefrag 中，是通过传 -Ttext 0x7C00 这个参数给链接程序设置了链接地址，因此链接程序在生成的代码中产生正确的内存地址。**Exercise 5**

修改 boot/Makefrag 让其加载地址出错。查看这个文件

$(OBJDIR)/boot/boot: $(BOOT\_OBJS)

@echo + ld boot/boot

$(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o $@.out $^

$(V)$(OBJDUMP) -S $@.out >$@.asm

$(V)$(OBJCOPY) -S -O binary -j .text $@.out $@

$(V)perl boot/sign.pl $(OBJDIR)/boot/boot

可以发现 -Ttext 后面的参数就是入口地址。如果把这个值修改为0x8C00，保存后回到lab1文件夹下进行make，查看 obj/boot/boot.asm 会发现，开头

Disassembly of section .text:

00008c00 <start>:

.set CR0\_PE\_ON, 0x1 # protected mode enable flag

.globl start

start:

.code16 # Assemble for 16-bit mode

cli # Disable interrupts

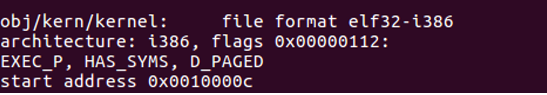
8c00: fa cli

cld # String operations increment

8c01: fc cld

可以发现起始地址从原来的 00007c00 变为 00008c00。虽然此时在0x7c00处打断点然后运行时正常的，但是继续si以后会在 [ 0:7c2d] => 0x7c2d: ljmp $0x8,$0x8c32 出循环，同时qemu端口出现了错误。因为不能ljmp到$0x7c32而是调到了$0x8c32，所以无法执行正确的指令。查看 boot.asm 可以知道上面这个指令是 ljmp $PROT\_MODE\_CSEG, $protcseg，是为了进入32位模式的。

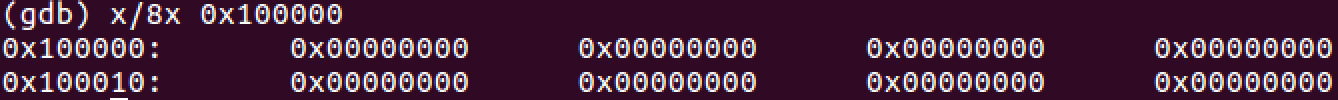
除了段信息，ELF头中的e\_entry字段也很重要。这个字段保存了程序入口点(entry point)的链接地址，也就是程序执行的text字段中的内存地址。使用一下命令查看objdump -f obj/kern/kernel

**Exercise 6**

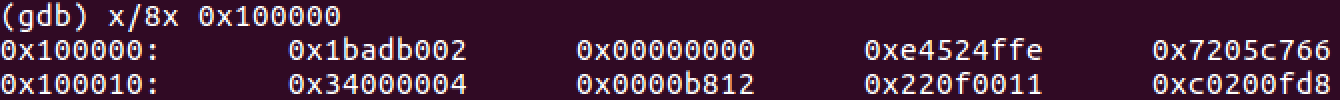
使用GDB的 x/Nx ADDR 可以打印内存地址ADDR的 N 个字。字的大小分情况的，GDB中一个字是两个字节。

查看BIOS启动时0x00100000处的8歌字，然后继续到boot loader进入内核的位置，再查看，发现8个字的内容不同。

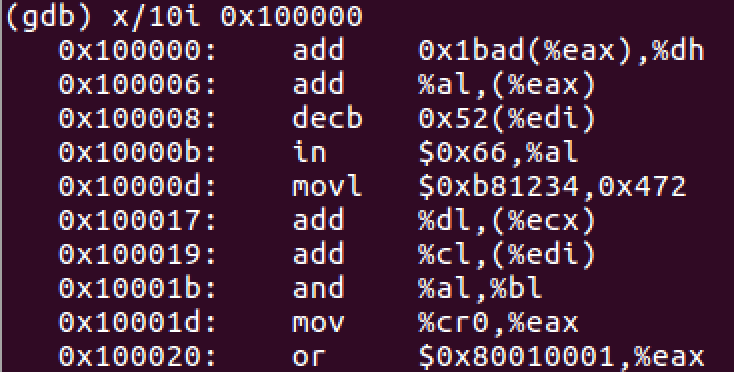
现在0x7c00处打断点，然后运行到断点处，使用 x/8x 0x100000 可以看到



根据之前的看到程序入口点是 0x10000c ，所以在 0x10000c 处打断点运行，同样可以看到



使用 x/10i 0x100000 会看到



应该能感觉到 0x100000 处存放的其实就是程序指令段，也就是说 bootmain 函数会把内核的程序段送到内存 0x100000 处。

* 1. **The Kernel**

**使用虚拟内存**

boot loader 的链接地址和加载地址是一样的，然而 kernel 的链接地址和加载地址有些差异。查看 kern/kernel.ld 可以发现内核地址在 0xF0100000。

操作系统内核通常被链接并且运行在非常高的虚拟地址，比如文件里看到的 0xf0100000，为了让处理器虚拟地址空间的低地址部分给用户程序使用。

许多机器没有地址为 0xf0100000 的物理内存，所以内核不能放在那儿。因此使用处理器内存管理硬件将虚拟地址 0xf0100000 (内核希望运行的链接地址)映射到物理地址 0x00100000 (boot loader加载内核后所放的物理地址)。尽管内核虚拟地址很高，但加载进物理地址位于1MB的地方仅仅高于BIOS的ROM。这需要PC至少有1MB的物理内存。

在下一个lab，会映射物理地址空间底部256MB，也就是 0x00000000 到 0x0fffffff，到虚拟地址 0xf0000000 ~ 0xffffffff。所以JOS只使用物理内存开始的256MB。

目前，只是映射了物理内存开始的4MB， 使用手写的静态初始化页目录和也表在 kern/entrypgdir.c。当 kern/entry.S 设置 CR0\_PG 标记，存储器引用就变为虚拟地址，即存储器引用是由虚拟存储器硬件转换为物理地址的虚拟地址。entry\_pgdir 将虚拟地址 0xf0000000 ~ 0xf0400000 转换为物理地址 0x00000000 ~ 0x00400000，虚拟地址 0x00000000 ~ 0x00400000 也转换为物理地址 0x00000000 ~ 0x00400000。任何不在这两个范围内的虚拟地址会导致硬件异常。

**Exercise 7**

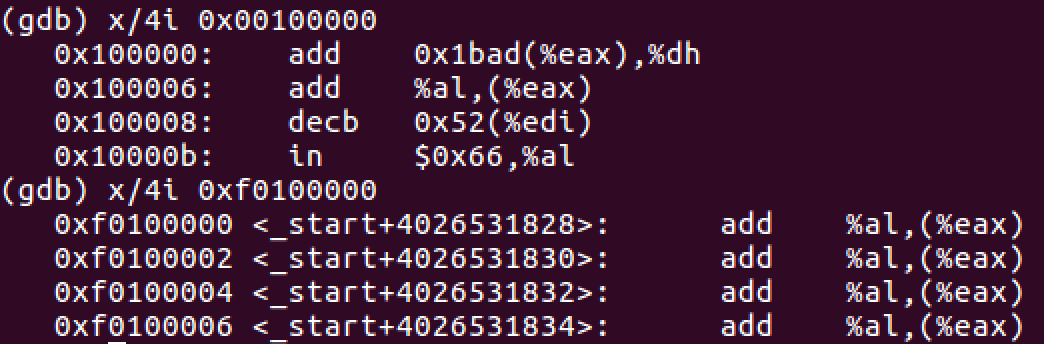
追踪JOS内核并停在 movl %eax, %cr0。查看内存 0x00100000 和 0xf0100000。接着使用 stepi 来看上面两个地址里内容的变化。

若注释了 kern/entry.S 的 movl %eax, %cr0, 查看第一个出现问题的指令是什么。

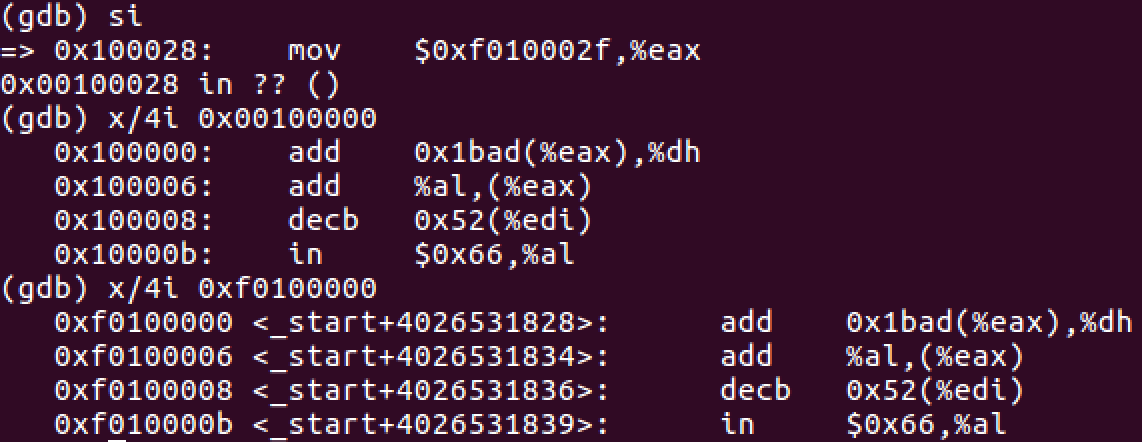
查看 kern/entry.S 发现 \_start 是ELF入口点，exercise 5 提到了入口点是 0x0010000c. 所以在0x0010000c处打断点。

(gdb) b \*0x0010000c

接着输入 c 使程序运行到断点处。使用 x/4i 来查看后四条指令,发现 0x00100000 和 0xf0100000 不同。

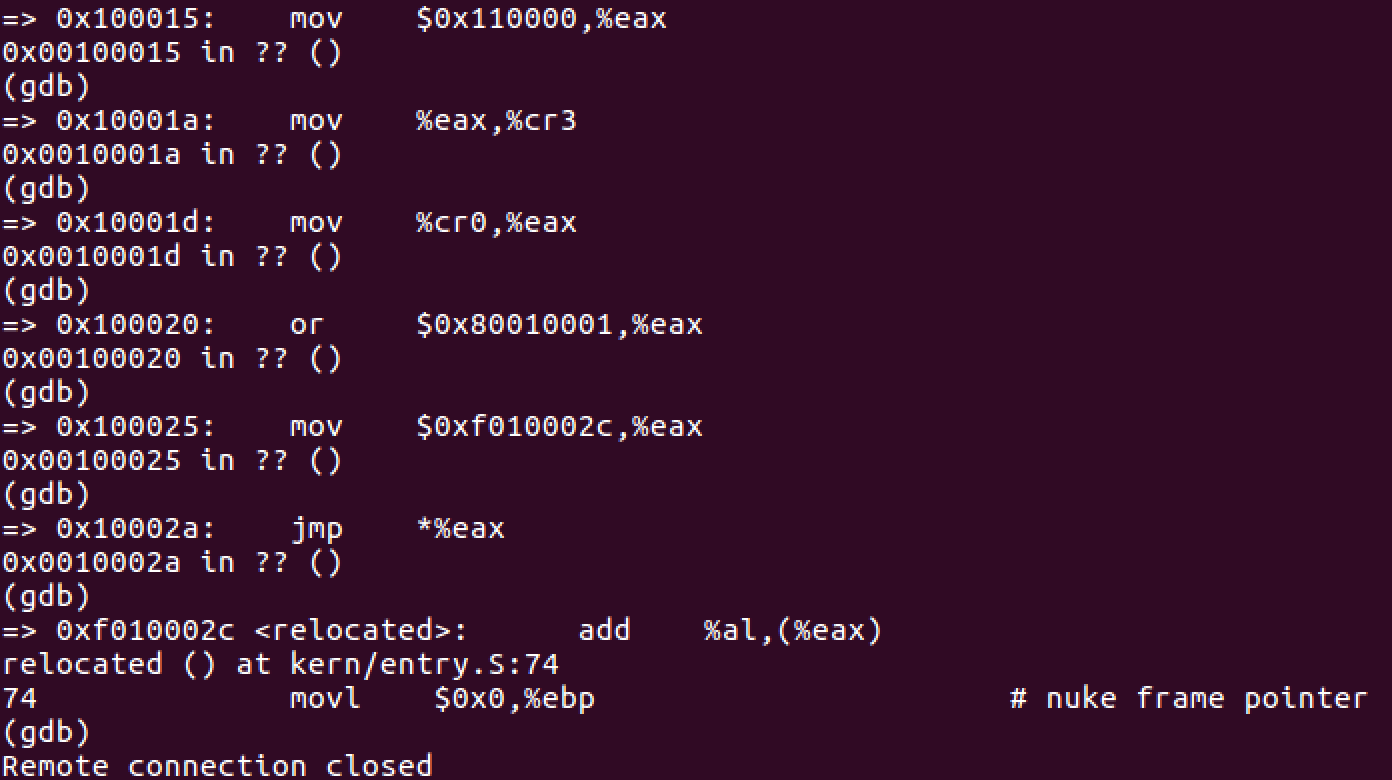


在执行完6次si后，终于 0x00100000 和 0xf0100000 处内容相同。

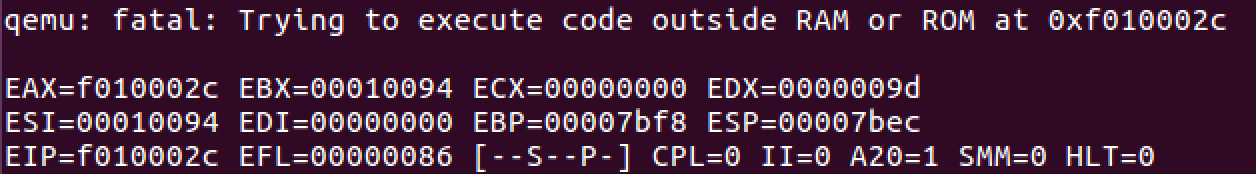


也就是说，0xf0100000 的内容被映射到 0x00100000。

注释 movl %eax, %cr0 后，make clean 之后重新编译，再运行。一步步 si 后出现了问题。



在0x10002a处的jmp指令，要跳到 0xf010002c 处， 然而因为没有分页管理，不会进行虚拟地址映射到物理地址的转化，再另一个窗口可以看到错误信息，访问地址超出内存。



### 格式化输出到控制台

分析 kern/printf.c, lib/printfmt.c, 和 kern/console.c 的代码。**Exercise 8**

完成指定输出”%o”格式字符串的代码。

首先分析 kern/printf.c, lib/printfmt.c, 和 kern/console.c 的关系。

kern/printf.c里的 vcprintf, cprintf 都调用 lib/printfmt.c 的 vprintfmt。kern/printf.c里的 putch调用 kern/console.c 的 cputchar。lib/printfmt.c 里也有 putch。

所以 kern/printf.c 和 lib/printfmt.c 依赖 kern/console.c。

首先先看看 putch 里调用的 cputchar。

// 'High'-level console I/O. Used by readline and cprintf.

void

cputchar(int c)

{

cons\_putc(c);

}

// output a character to the console

static void

cons\_putc(int c)

{

serial\_putc(c);

lpt\_putc(c);

cga\_putc(c);

}

cputchar调用的是 cons\_putc, 所以 cons\_putc 才是关键。 cons\_putc 的功能是输出一个字符到控制台。cons\_putc 又是由 serial\_putc， lpt\_putc 和 cga\_putc 组成。

因此，要先来看 serial\_putc。

#define COM1 0x3F8

#define COM\_LSR 5 // In: Line Status Register

#define COM\_LSR\_TXRDY 0x20 // Transmit buffer avail

#define COM\_TX 0 // Out: Transmit buffer (DLAB=0)

static void

serial\_putc(int c)

{

int i;

for (i = 0;

!(inb(COM1 + COM\_LSR) & COM\_LSR\_TXRDY) && i < 12800;

i++)

delay();

outb(COM1 + COM\_TX, c);

}

它控制的是端口0x3F8，inb 读取的是 COM1 + COM\_LSR = 0x3FD 端口，outb 输出到了 COM1 + COM\_TX = 0x3F8。

详细端口信息查看这个[PORTS.LST](http://bochs.sourceforge.net/techspec/PORTS.LST)。

在 inb(COM1 + COM\_LSR) 之后， 有 & COM\_LSR\_TXRDY 这个操作。 !(inb(COM1 + COM\_LSR) & COM\_LSR\_TXRDY)其实是为了查看读入的数据的第6位，也就是 PORTS.LST 中 03FD 中提到的 bit 5 是否为1。 如果为1，上面的语句结果就是0，停止for循环。这个 bit 5 是判断发送数据缓冲寄存器是否为空。

outb 是将端口 0x3F8 的内容输出到 c。当 0x3F8 被写入数据，它作为发送数据缓冲寄存器，数据是要发给串口。

所以serial\_putc是为了把一个字符输出到串口。

再来看 lpt\_putc。

/\*\*\*\*\* Parallel port output code \*\*\*\*\*/

// For information on PC parallel port programming, see the class References

// page.

static void

lpt\_putc(int c)

{

int i;

for (i = 0; !(inb(0x378+1) & 0x80) && i < 12800; i++)

delay();

outb(0x378+0, c);

outb(0x378+2, 0x08|0x04|0x01);

outb(0x378+2, 0x08);

}

将字符给并口设备。

最后一个是 cga\_putc。

static void

cga\_putc(int c)

{

// if no attribute given, then use black on white

if (!(c & ~0xFF))

c |= 0x0700;

switch (c & 0xff) {

case '\b':

if (crt\_pos > 0) {

crt\_pos--;

crt\_buf[crt\_pos] = (c & ~0xff) | ' ';

}

break;

case '\n':

crt\_pos += CRT\_COLS;

/\* fallthru \*/

case '\r':

crt\_pos -= (crt\_pos % CRT\_COLS);

break;

case '\t':

cons\_putc(' ');

cons\_putc(' ');

cons\_putc(' ');

cons\_putc(' ');

cons\_putc(' ');

break;

default:

crt\_buf[crt\_pos++] = c; /\* write the character \*/

break;

}

// What is the purpose of this?

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;

}

/\* move that little blinky thing \*/

outb(addr\_6845, 14);

outb(addr\_6845 + 1, crt\_pos >> 8);

outb(addr\_6845, 15);

outb(addr\_6845 + 1, crt\_pos);

}

首先 !(c & ~0xFF) 是否在 0 ~ 255 之前。\b很容易理解，就是退格键，让缓冲区 crt\_buf 的下标 crt\_pos 减1。其他的同理，case都是格式操作。default就是往缓冲区里写入字符c。之后就是当缓存超过CRT\_SIZE，就是用 memmove 复制内存内容。

最后四句代码是将缓冲区的内容输出到显示屏。

### 内联汇编tips

这里提一下内联汇编。 inb 和 outb 都是内联汇编。其中 inb 函数

inb(int port)

{

uint8\_t data;

\_\_asm \_\_volatile("inb %w1,%0" : "=a" (data) : "d" (port));

return data;

}

“\_\_asm\_\_“表示后面的代码为内联汇编， “\_\_volatile\_\_“表示编译器不要优化代码。括号里是汇编指令。

内联汇编的模板是：\_\_asm\_\_(汇编语句模板: 输出部分: 输入部分: 破坏描述部分)。 共四个部分：汇编语句模板，输出部分，输入部分，破坏描述部分，各部分使用”:”格开，汇编语句模板必不可少， 其他三部分可选，如果使用了后面的部分，而前面部分为空，也需要用”:”格开，相应部分内容为空。如上面程序则只是用了前三部分。

“inb %w1,%0”表示汇编语句模板， “=a” (data)是输出部分，”d” (port)是输入部分。本例中只有两个：“data” 和“port”，他们按照出现的顺序分别与指令操作数 “%0”,“%1”对应。每个输出操作数的限定字符串必须包含“=”表示他是一个输出操作数。例如”=a” (data)就是一个输出操作数，其限定字符串为”=a”，(data)就是C语言变量。

之后来看 lib/printfmt.c 的代码。首先来看 kern/printf.c 里提到的 vprintfmt函数。

代码太长了，简单点先说一下它的四个输入参数。

* void (\*putch)(int, void\*) 函数指针，一般调用输出到屏幕上的函数
* void \*putdat 输入字符要放的内存地址指针
* const char \*fmt 格式化字符串
* va\_list ap 多个输入参数

在vprintf里传入的参数putdat是cnt的地址，cnt是用来做计数器的。cprintf功能类似。

分析完三个文件，回到题目，要去实现%o的格式化输出。在 lib/printfmt.c 可以看到要填写的地方。参考上面 case 'u' 的写法。

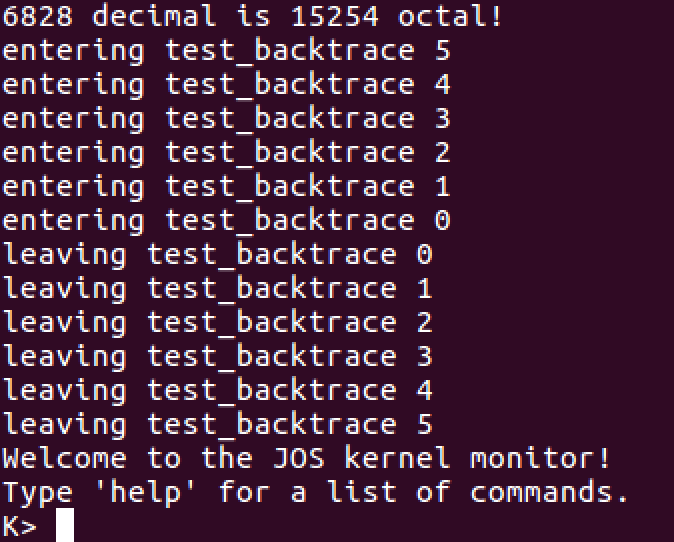
case 'o':

num = getuint(&ap, lflag);

base = 8;

goto number;

修改完以后保存，make clean 之后运行，会发现启动以后，qemu里JOS启动时会出现这样一行字。



第一行完成了6828转八进制。

解释 console.c 中的这段代码

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;

}

简单点说就是当字符长度超过CRT\_SIZE， 证明屏幕放不下了，需要页面向上滚动一行。

观察下面的代码，分析fmt和ap指向什么。

int x = 1, y = 3, z = 4;

cprintf("x %d, y %x, z %d\n", x, y, z);

fmt 指向的是格式字符串 “x %d, y %x, z %d\n”，ap 指向的是参数 x, y, z。  
将这段代码加入到 kern\monitor.c 并重新编译运行，即可显示结果。

运行这段代码。

unsigned int i = 0x00646c72;

cprintf("H%x Wo%s", 57616, &i);

输出是 He110 World。 这很神奇。首先 %x 是指十六进制，所以将 57616 转为十六进制就是 e110。在看后面， &i 是指 i 的地址， %s是指输出为字符串，所以输出的应该是变量i所在地址的字符串。其实就是将i转换为字符串来输出。因为x86是小端模式，并且是int类型，所以存放的方式是四个字节，所以就会将i拆分开来，变成 0x72, 0x6c, 0x64, 0x00. 所以转换为字符就是rld\0，所以得到了上面的输出结果。

运行以下代码。

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

结果是 x=3 y=-267380708，y出问题是因为没有被指定值，所以输出的是一个不确定的值。

### 堆栈

最后这部分，要研究C语言是如何在x86框架上使用堆栈的。需要查看指令寄存器(IP)的值的变化。

**Exercise 9**

研究内核是在哪初始化堆栈，找出堆栈存放在内存的位置。内核是如何保存一块空间给堆栈的？堆栈指针指向这块区域的哪儿？

看了几个文件以后，发现在 kern/entry.S 中提到了设置堆指针和栈指针。

# Clear the frame pointer register (EBP)

# so that once we get into debugging C code,

# stack backtraces will be terminated properly.

movl $0x0,%ebp # nuke frame pointer

# Set the stack pointer

movl $(bootstacktop),%esp

为了查看堆的位置，所以要使用gdb，同样还是 b \*0x10000c 打断点进入 entry。 si 一步步执行，在 0x10002d: jmp \*%eax 之后，下一条指令变为 0xf010002f <relocated>: mov $0x0,%ebp。其实地址应该还是 0x10002f，所以这里的 0xf010002f 是因为开启的虚拟地址。

通过 gdb 发现 0xf0100034 <relocated+5>: mov $0xf0110000,%esp， 也就是说%esp也就是bootstacktop的值为0xf0110000。其中 kern/entry.S 的 KSTKSIZE 应该就是堆栈的大小，通过跳转，发现在 inc/memlayout.h 里提到了堆栈。

// Kernel stack.

#define KSTACKTOP KERNBASE

#define KSTKSIZE (8\*PGSIZE) // size of a kernel stack

#define KSTKGAP (8\*PGSIZE) // size of a kernel stack guard

PGSIZE 定义在 inc/mmu.h 中，值为 4096，所以 KSTKSIZE 为 32KB。 使用 info registers 可以查出esp和ebp的值。最高地址为bootstacktop的值，也就是0xf0110000。

x86堆栈指针(esp寄存器)指向堆栈正在使用的最低位置，低于这个位置的空间还没使用。ebp寄存器(基址指针寄存器)与程序有关。详细的内容可以看 CSAPP。

**Exercise 10**

研究 obj/kern/kernel.asm 中 test\_backtrace 向堆栈里压入的信息。

// Test the stack backtrace function (lab 1 only)

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);

}

使用单步调试和 info registers 来查看esp和ebp的变化。

运行test\_backtrace前，

esp 0xf010ffdc 0xf010ffdc

ebp 0xf010fff8 0xf010fff8

执行的操作为

0xf0100040 <test\_backtrace>: push %ebp #栈底指针ebp压入栈中

0xf0100041 <test\_backtrace+1>: mov %esp,%ebp #将栈顶指针esp指向栈底指针ebp

0xf0100043 <test\_backtrace+3>: push %ebx #压入ebx基底寄存器

0xf0100044 <test\_backtrace+4>: sub $0xc,%esp #栈顶指针esp向低地址移动0xc也就是12bytes。

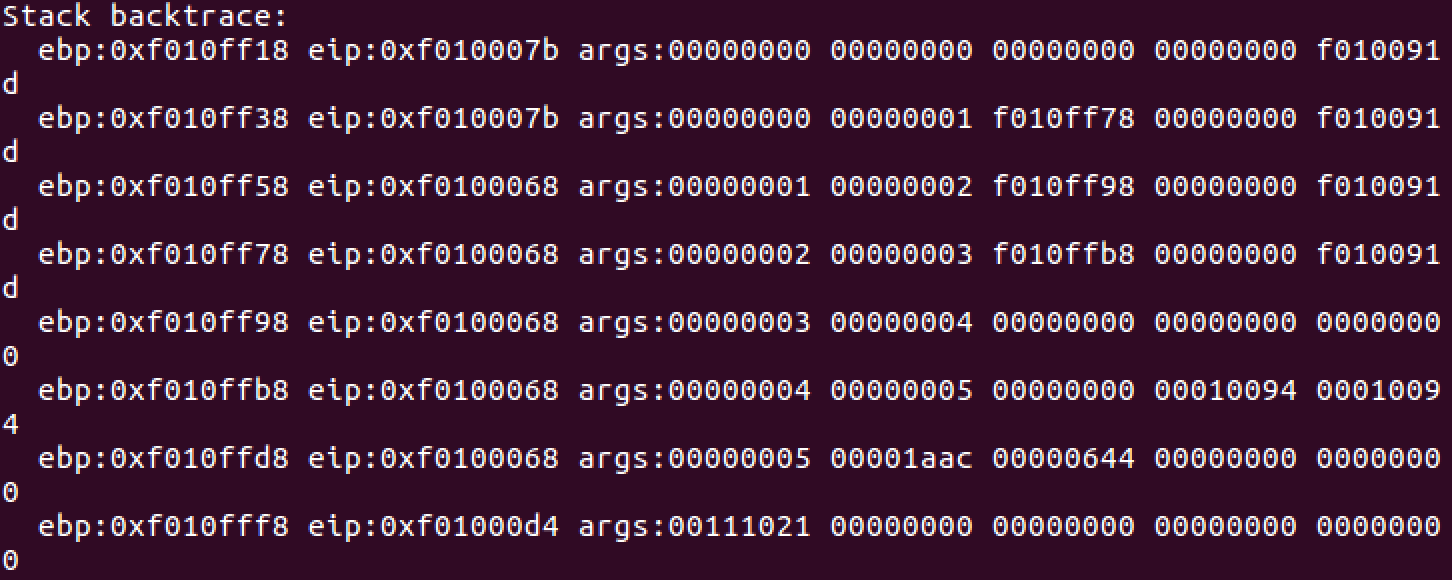
0xf0100047 <test\_backtrace+7>: mov 0x8(%ebp),%ebx

0xf010004a: 53 push %ebx #将%ebp+8位置的变量压入栈中，为之后的调用做准备。

总之，这段程序算是递归，递归就是ebp存放返回地址，esp是存放调用时的参数。

现在需要实现mon\_backtrace()这个函数，需要显示ebp，eip 和 args。ebp是基址指针，eip是返回指令指针。

简单实现backtrace，实现效果如下



实现如下。

int

mon\_backtrace(int argc, char \*\*argv, struct Trapframe \*tf)

{

// Your code here.

int j;

uint32\_t ebp = read\_ebp();

uint32\_t eip = \*((uint32\_t \*)ebp+1);

cprintf("Stack backtrace:\n");

while ((int)ebp != 0)

{

cprintf(" ebp:0x%08x eip:0x%08x args:", ebp, eip);

uint32\_t \*args = (uint32\_t \*)ebp + 2;

for (j = 0; j < 5; j ++) {

cprintf("%08x ", args[j]);

}

cprintf("\n");

eip = ((uint32\_t \*)ebp)[1];

ebp = ((uint32\_t \*)ebp)[0];

}

return 0;

}

**Exercise 11:**

修改上面实现的backtrace，要显示详细的函数地址。可以使用 kern/kdebug.c 的 debuginfo\_eip()。在查看debuginfo\_eip时发现其中有一段代码需要填写。这段代码是填写eip\_line。这里用到了写好的二分查找。

// Search within [lline, rline] for the line number stab.

// If found, set info->eip\_line to the right line number.

// If not found, return -1.

//

// Hint:

// There's a particular stabs type used for line numbers.

// Look at the STABS documentation and <inc/stab.h> to find

// which one.

// Your code here.

stab\_binsearch(stabs, &lline, &rline, N\_SLINE, addr);

if(lline <= rline){

info->eip\_line = stabs[rline].n\_desc;

}

else

info->eip\_line = -1;

关于 stab 相关内容，查看第三个参考资料，实话说，我对于stab理解的比较模糊。

**Exercise 12**

将backtrace嵌入终端中，使其可以被调用。只需要修改 kern/monitor.c 的

static struct Command commands[] = {

{ "help", "Display this list of commands", mon\_help },

{ "kerninfo", "Display information about the kernel", mon\_kerninfo },

{ "backtrace", "Display a listing of function call frames", mon\_backtrace}

};

### 实验结果

## Lab 2: Memory Management

### 1.实验目的

### 2.实验内容（附过程截图）

### 3.实验结果

## Lab 3: User Environments:

### 1.实验目的

### 2.实验内容（附过程截图）

### 3.实验结果

# 四、xv6实验系统分析

## xv6的系统结构（结合示意图）

## （二）xv6的引导

## （三）xv6的进程与调度分析

## （四）xv6的内存管理

## （五）xv6的文件系统

## （六）xv6 的I/O

处理器必须像和主存交互一样同设备交互。x86 处理提供了特殊的 in, out 指令来在设备地址（称为'I/O 端口'） 上读写。这两个指令的硬件实现本质上和读写内存是相同的。早期的 x86 处理器有一条附加的地址线：0表示从 I/O 端口读写，1则表示从主存读写。每个硬件设备会处理它所在 I/O 端口所接收到的读写操作。设备的端口使得软件可以配置设备，检查状态，使用设备；例如，软件可以通过对 I/O 端口的读写，使磁盘接口硬件对磁盘扇区进行读写。

很多计算机体系结构都没有单独的设备访问指令，取而代之的是让设备拥有固定的内存地址，然后通过内存读写实现设备读写。实际上现代 x86 体系结构就在大多数高速设备上（如网络、磁盘、显卡控制器）使用了该技术，叫做 内存映射 I/O。但由于向前兼容的原因， in, out 指令仍能使用，而比较老的设备如 xv6 中使用的 IDE 磁盘控制器仍使用两个指令。

首先，我们从**用户模式**介绍xv6.

I/O 系统调用部分

1. 使用系统调用的文件描述符

read (int fd, char\* buf, int len);

write (int fd, char\* buf, int len);

stat (int fd, struct stat \* );

dup (int fd);

close (int fd);

1. 命名服务

link (char ∗ oldpath , char ∗ newpath);

fd = open (char ∗ path , int flags);

unlink (char ∗ path);

mkdir (char ∗ path );

rmdir (char ∗ path );

还有管道函数用作文件缓冲

pipe (int pipefd[2]);

然后，我们开始进入xv6的核心

xv6核心是一个分层系统，其中文件层由pipe子系统和inode子系统组成，inode子系统包括name层、inode层、buffer层和driver层， 系统调用调用文件层，name层和inode层。

简单的介绍后，我们来探讨一下xv6如何在不同的硬件，结构和函数方法下，向用户模式提供统一的界面。

xv6 文件的抽象格式如下：

struct file{

enum {FD\_NONE, FD\_PIPE , FD\_INODE} type;

int ref; // reference count

char readable;

char writeable;

struct pipe∗ pipe;

struct inode ∗ ip;

uint off;};

其中各个变量功能下文均有解释，并且均将细化到代码模块

**I/O 子系统**

由以下几个模块组成

FD\_PIPE.

FD\_INODE:

T\_FILE, T\_DIR, T\_DEV.

每个 I/O 子系统都是由一个结构体和一个操作集合来定义的，子系统中的各个模块将在下文予以解释

FD\_PIPE模块

struct pipe{

struct spinlock lock;

char data[PIPESIZE];

uint nread;//读取的字符数目

uint nwrite;//写入的字符数目

intreadopen;//读取的文件仍然开启

intwriteopen;/写入的文件仍然开启

};

还有pipe的一些相关基本操作

pipalloc, pipeclose, piperead, pipewrite.

FD\_INODE模块

struct inode{

uint dev;//设备数字

uint inum;//Inode数字

int ref;//Reference count

int flags;//IBUSY,IVALID

short type;// inode的磁盘镜像

short major;

short minor;

short nlink;

uint size;

uint addrs[NDIRECT+1];

};

F\_INODE相关方法

Listing1:读入T\_FILE or T\_DIR

调用如下方法

namei, create, ilock, readi, writei, stati, iunlock, iput

Listing2:读入T\_DEV

consoleread, consolewrite

**根据文件层的系统调用**

sys dup, sys read, sys write, sys fstat, sys close.

在fd系统调用中使用的函数

struct file ∗ filedup(struct file ∗f)

fileclose(struct file ∗f);

fileread(struct file ∗f, char ∗buf, int len);

filewrite(struct file ∗f,char∗ buf, int len);

filestat(struct file∗f, struct stat∗s);

fdalloc:用于返回最低限度的被数字标记的空余文件槽.

static int fdalloc(struct file ∗f){

int fd;

for(fd = 0;fd < NOFILE; fd ++){

if(proc−>ofile[fd]==0){

proc−>ofile[fd] = f;

return fd;

}

}

return −1;

}

sys\_dup部分

sys\_dup(void){

struct file ∗f;

int fd;

if(argfd(0,0,&f)<0) return −1;

if((fd = fdalloc(f))<0) return −1;

filedup(f);

return fd;

}

sys\_read部分

int sys\_read(void){

struct file ∗f;

int n;

char ∗p;

if(argfd(0,0,&f)<0||argint(2,&n)<0||argptr(1,&p,n)<0) return −1;

return fileread(f,p,n);

}

sys\_write部分

int sys\_write(void){

struct file ∗f;

int n;

char ∗p;

if(argfd(0,0,&f)<0||argint(2,&n)<0||argptr(1,&p,n)<0) return −1;

return filewrite(f,p,n);

}

sys\_fstat部分

int sys\_fstat(void){

struct file ∗f;

struct stat ∗st;

if(argfd(0,0,&f)<0||argptr(1,(void∗)&st,sizeof(∗st))<0) return−1;

return filestat(f,st);

}

sys\_close部分

int sys\_close(void){

int fd;

struct file∗f;

if(argfd(0,&fd,&f)<0) return −1;

proc−>ofile[fd]=0;

fileclose(f);

return 0;

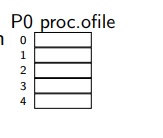
}

文件层的实现通过以下五个函数:

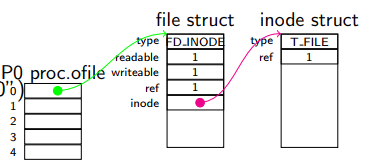
filedup，fileread，filewrite，filestat，fileclose.

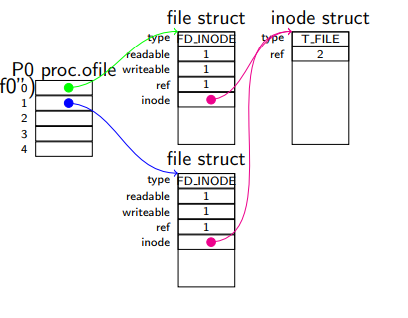
我们先来看一下ofile/file/inode/pipe 的相关结构

1. 无文件开启时

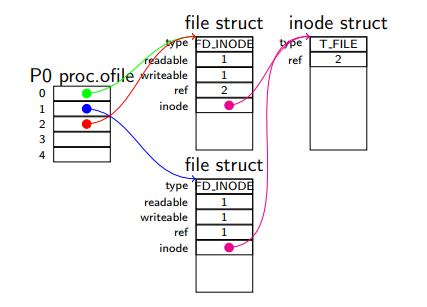


1. 执行open(”/carmi/f0”)

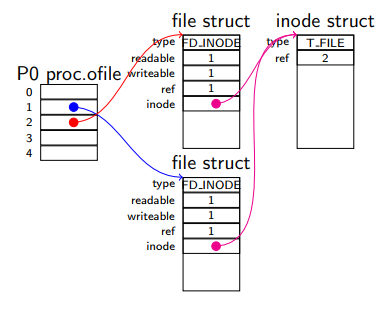




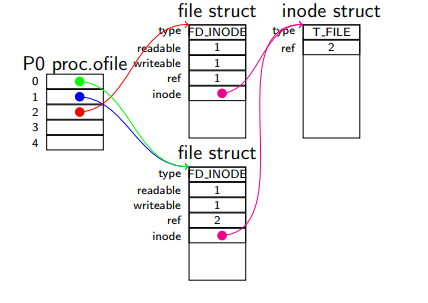
1. 执行dup(0)



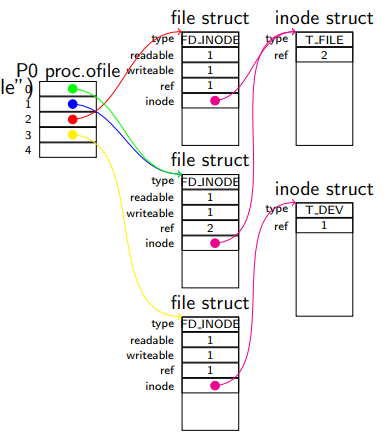
1. 执行close(0)



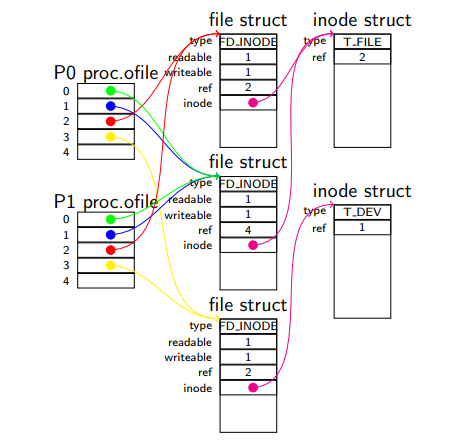
1. 执行dup(1)



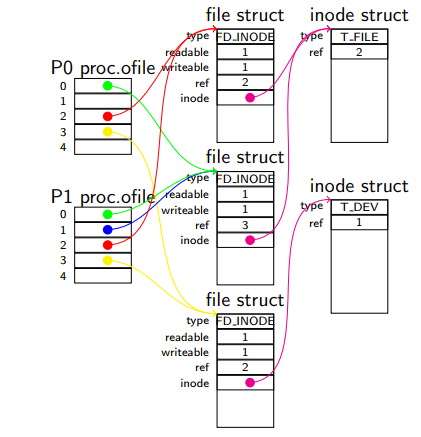
1. 执行open(”/console”)



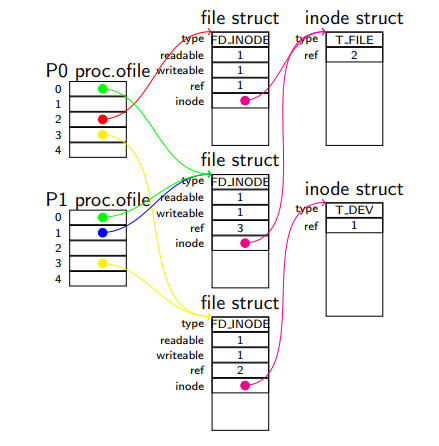
1. 执行fork()



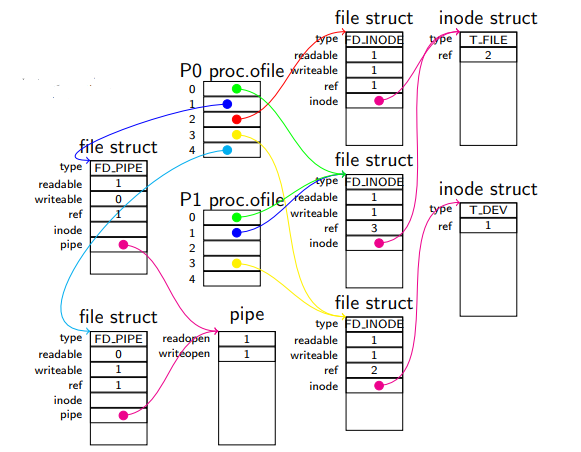
1. 执行P0 close(1)



1. 执行P1 close(2)



1. 执行P0 pipe()



**文件层调度**

文件层将调度一下子系统之一:

1.pipe.

2.inode.

pipe子系统和inode层使用的函数

Pipe子系统:

•piperead(pipe \*p, char \*addr, int len);

•pipewrite(pipe \*p ,char \*addr, int len);

•pipeclose(pipe \*p ,int writeside);

inode子系统:

•ilock(inode \*ip);

•iunlock(inode \*ip);

•readi(inode \*ip, char \*adr, int len);

•writei(inode \*ip, char\* adr, int len);

•iput(inode \*ip);

•stati(inode \*ip, stat\* stat);

•begintrans();

•committrans();

模块详细说明

file\_dup模块

struct file∗ file\_dup(struct file∗f){

acquire(&ftable.lock);

if(f−>ref<1) panic(”filedup”);

f−>ref++;

release(&ftable.lock);

return f;

}

根据类型，file\_read将读取委托给以下其中一个：

管道,readi。

对于FD\_INODE类型，处理文件位置。

file\_read模块

int file\_read(structfile ∗f, char ∗addr,int n){

int r;

if(f−>readable==0) return −1;

if(f−>type==FDPIPE) return pipe\_read(f−>pipe,addr,n);

if(f−>type==FDINODE){

ilock(f−>ip);

if((r=readi(f−>ip,addr,f−>off,n))>0) f−>off+=r;

iunlock(f−>ip);

return r;

}

panic(”fileread”);

}

根据类型，filewrite将写入委托给以下其中一个：

pipe写入，writei。

对于FD\_INODE类型，处理文件位置。

file\_write模块

int file\_write(struct file ∗f, char∗ addr, int n){

if(f−>writable==0) return−1;

if(f−>type==FDPIPE) return pipe\_write(f−>pipe,addr,n);

if(f−>type==FDINODE){

int max=((LOGSIZE−1−1−2)/2)∗512;

for(int i=0;i<n;){

int n1=n−i;

if(n1>max) n1=max;

begintrans();

ilock(f−>ip);

if((r=writei(f−>ip,addr+i,f−>off,n1))>0) f−>off+=r;

iunlock(f−>ip);

committrans();

if(r<0) break;

if(r!=n1) panic(”shortfilewrite”);

i+=r;

}

return i==n ? n:−1;

}

panic(”filewrite”);

}

reference count已更新。

如果reference count下降到零，我们委托给其中一个：

pipe关闭, iput。

由于iput我们必须释放需要注意，因此需要

复制文件结构。

file\_stat模块

int file\_stat(structfile ∗f,structstat ∗st){

if(f−>type==FDINODE){

ilock(f−>ip);

stati(f−>ip,st);

iunlock(f−>ip);

return 0;

}

return −1;

}

file\_close模块

void file\_close(struct file ∗f){

acquire(&ftable.lock);

if(f−>ref<1) panic(”fileclose”);

if(−−f−>ref>0){

release(&ftable.lock);

return;

}

structff=∗f;

f−>ref=0;

f−>type=FDNONE;

release(&ftable.lock);

if(ff.type==FDPIPE) pipeclose(ff.pipe,ff.writable);

elseif(ff.type==FDINODE){

begintrans();iput(ff.ip);committrans();

}

}

文件层基本实现

pipe 的逻辑：委托给pipealloc，隐藏返回的file指针到ofile向量里.

sys\_pipe模块

int sys\_pipe(void){

int∗ fd;

structfile ∗rf,∗wf;

int fd1;

if(argptr(0,(void∗)&fd,2∗sizeof(fd[0]))<0) return −1;

if(pipealloc(&rf,&wf)<0) return −1;

int fd0=−1;

if((fd0=fdalloc(rf))<0||(fd1=fdalloc(wf))<0){

if(fd0>=0) proc−>ofile[fd0]=0;

fileclose(rf);

fileclose(wf);

return −1;

}

fd[0]=fd0;

fd[1]=fd1;

ret

# 五、总结与心得

（做了这些实验之后有哪些收获；觉得MIT-XV6系统有哪些地方值得改进；对操作系统实验有哪些建议）