Lab1:Booting a PC

介绍

这个实验被分为三个部分:

- 熟悉X86汇编语言, QEMU x86 模拟器, PC 开机的引导程序。
- 测试6.828内核的引导加载程序 (位于lab文件夹中的boot文件夹)
- 钻研6.828内核本身的初始化模板, 也叫 `JOS`

软件设置

- 安装 `git` 工具
- 安装 `qemu` 和可能的 `gcc` 工具

提交进度

去该网站申请一个 `API key`

输入 `make handin` , 按照提示完成操作。

Part1: PC 引导程序

第一个练习的目的是向你介绍x86汇编语言, PC引导程序, 和让你使用 `QEMU` && `QEMU/GDB` debug来开始实验。

在这个部分你不用写任何代码, 但是要**自己理解**并准好回答下面的问题! 😃

从x86汇编语言开始

如果你还不熟悉x86汇编语言, 你将在这个课程中熟悉它。

参考书籍: PC Assembly Language Book

warning: 该书用的是 *Intel* 语法, 而GNU使用的是 *AT&T* 语法。二者的转换在Brennan's Guide to Inline Assembly 中有介绍。

For Brennan's Guide to Inline Assembly

1. 基本语法

   和 *csapp* 中的汇编部分基本相同，比较简单。

2. 基本内置汇编

```
asm("statements");
asm("nop");
```

   如果 `asm` 和程序中的命名有冲突，也可以使用 `__asm__`

```
asm("push %eax\n\t"
    "movl $0, %eax\n\t")
    "popl %eax");         // \n\t 需要使用如果是多条汇编语句
```

   如果要修改寄存器中的内容，要使用下面扩展的内置汇编

3. 扩展的内置汇编

   basic format

```
asm ( "statements" : output_registers : input_registers :
clobbered_register);
    // 语句：输出寄存器：输入寄存器：被修改的寄存器
```

   a nifty example

```
asm ( "cld\n\t"      // 将标志寄存器中的方向标志位清除 决定内存地址是增
大（DF=0）还是减小
      "rep\n\t"      // 重复执行后面的指令，重复次数位ecx中的值
      "stosl"        // 将eax中的数据赋值给后面的地址
      : /* no output registers */
      : "C" (cout), "a" (fill_value), "D" (dest)
                               // 将count放入ecx，fill_value放入
eax,目的地址放入edi
      : "%ecx", "%edi"  );
```

   如果要修改一个变量的话，记得在 `clobber list` 添加 `memory` 。

   下面是一些字母所代表的寄存器

```
a        eax
b        ebx
c        ecx
d        edx
S        esi
D        edi
I        constant value (0 to 31)
q,r      dynamically allocated register (see below)
g        eax, ebx, ecx, edx or variable in memory
A        eax and edx combined into a 64-bit integer (use long
longs)
```

example

```
asm ("leal (%1,%1,4), %0"         // 在生成过程中0%将会被GCC替换为
一个可用的
     : "=r" (x)                   // 'r'使得GCC考虑使用esi和edi寄
存器
     : "0" (x) );
```

=用来特指一个 输出寄存器

一般情况下，如果不是 `rep movsl` 、 `rep stosl` 这种明确要使用到 `ecx` ， `edi` ， `esi` 寄存器的汇编指令，我们可以让编译器GCC来选择一个可以使用的。

不需要将GCC分配的寄存器放入clobberlist。

指定特定寄存器需要用两个%

```
asm ("leal (%%ebx,%%ebx,4), %%ebx"
     : "=b" (x)
     : "b" (x) );
```

注意

如果你的汇编语句必须在输入的地方执行（禁止被移出循环作为优化），请使用关键字 `volatile`

```
__asm__ __volatile__( …… )
```

如果目的仅仅是计算输出寄存器，并没有其他影响，你最好不使用 `volatile` 关键字。

参考链接：https://www.cnblogs.com/zhuyp1015/archive/2012/05/01/2478099.html

一些将来可能会有用的使用者手册

- 6.828 reference page
- old 80386 Programmer's Reference Manual

  更简短，更容易指导我们去使用，但是描述了所有我们在6.828中将会使用到的x86处理器特性
- IA-32 Intel Architecture Software Develop‘s Manuals

  覆盖了因特尔所有最近处理器的特性
- AMD

  通常更友好。

模拟x86处理器

在6.828中我们将使用*QEMU* 模拟器，该模拟器可以作为一个远程调试目标（为了GDB），在步入早期的boot进程中我们会使用到GDB。

输入 `make qemu` 或者 `make qemu-nox` 后， `k>` 就是交互的控制程序。

如果使用的是 `make qemu` ,这些命令行不仅会在常见的shell窗口出现也会在QEMU窗口出现。

可以通过输入 `make qemu-nox` 来只通过命令行来使用，如果你说用ssh登陆的话，这可能对你更方便。

退出 `qemu` ，按 `ctrl+a x`

command list

对于kernel monitor，只有两条命令。

- `help`
- `kerninfo`

PC物理地址空间

```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

从0x000A0000 到 0x000FFFFF 是被硬件保留作为特殊用途的空间，比如视频播放缓存。

这部分最重要的是 Basic Input/Output System，占据了从0x000F0000到0x000FFFFF的内存空间。

- 早期的PC，BIOS存储于只读存储器（ROM），但是现在的PC将其存储于可升级的闪存中。
- BIOS负责对基本系统初始化，比如激活显卡（Video card），检查安装的内存大小。
- 初始化之后，BIOS会从合适的地方加载操作系统并向操作系统传递对机器的控制。

自从intel用80286和80386处理器打破了物理内存兆字节的屏障后，PC的架构不再保持原来为了确保和现存的软件向后兼容的低1MB原始布局。

现代PC有一个“hole”，从0x000A0000 到 0x00100000 ，将RAM分为低内存和扩展内存。

作为补充，在PC的32位地址空间的最顶部，所有物理RAM的上面，一般被BIOS保存，为了32位PCI设备的使用。

最近的x86处理器可以支持多于4GB的物理RAM。

这种情况下，BIOS必须安排在32位可寻址区域留下 *第二个* “hole”，以便为那些32位设备留下空间去映射。

由于设计的限制，JOS将仅使用PC物理内存的前256MB，所以现在我们将假设所有的PC只有32位的物理地址空间。

The ROM BIOS

在这部分，你将会使用QEMU的debug工具来探索一个兼容IA-32电脑如何启动。

- 打开两个终端窗口，进入lab文件夹，一个终端窗口输入 `make qemu-nox-gdb` ，另一个输入 `make gdb` 。

  `make gdb` 输入前要确保已经安装了 `GDB` 。

  ```
  sudo apt-get install -y build-essential gdb
  ```

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

  这是GDB第一条被执行的汇编指令。从这个输出我们可以总结一些东西：

  1. IBM PC从物理内存地址 0x000ffff0 开始执行，这个地址是为ROM BIOS 保存的64KB区域的最上面。
  2. PC从 `cs = 0xf000` 和 `ip = 0xfff0` 开始执行。
  3. 第一个被执行的是 `jmp` 指令，跳转到段地址 `cs = 0xf000` 和 `IP = 0xe05b`

  为什么QEMU会这么开始？这就是Intel如何设计的8088处理器，IBM也在他们的原始PC上使用。

  因为BIOS在PC中在 0x000f0000-0x000fffff 地址范围内是“硬连线”的，这样的设计是为了确保BIOS总是能在启动后第一时间控制机器。

  这很重要，因为启动时机器的RAM里面没有任何其他可以执行的软件。

  QEMU模拟器自带它自己的BIOS，被放置在处理器的模拟物理地址空间。当处理器复位时，模拟处理器进入实模式并且设置 `cs` 为0xf000 和 `IP` 为0xfff0，因此执行将从此段地址（ `CS:IP` ）开始。
  ```

如何将 `0xf000:fff0` 转换成物理地址呢?

- 在实模式中（PC启动的模式），地址解释根据一个公式： *物理地址 = 16 ✕ 段地址 + 偏移地址* 。
- 所以当PC设置 `cs = 0xf000` 和 `IP = 0xfff0` ，物理地址计算如下：

  ```
  16 * 0xf000 + 0xfff0 = 0xf0000 + 0xfff0 = 0xffff0
  ```

`0xffff0` 在BIOS结束地址（0x100000）之前16个字节。我们不应该惊讶于BIOS做的第一件事情是跳转回一个更之前的位置；

毕竟它能在这16个字节完成多少呢?

Exercise

> 用GDB的 `si` 命令，跟踪进入ROM BIOS以获取更多的指令，并试着去猜测它可能正在做什么。
>
> Preference:Phil Storrs I/O Ports Description
>
> ```
> [f000:e05b]    0xfe05b:
>                          cmpl   $0x0,%cs:0x6ac8
> (gdb) si
> [f000:e062]    0xfe062:
>                          jne    0xfd2e1
> 0x0000e062 in ?? ()
> (gdb) si
> [f000:e066]    0xfe066:
>                          xor    %dx,%dx
> 0x0000e066 in ?? ()
> (gdb) si
> [f000:e068]    0xfe068:
>                          mov    %dx,%ss
> 0x0000e068 in ?? ()
> (gdb) si
> [f000:e06a]    0xfe06a:
>                          mov    $0x7000,%esp
> 0x0000e06a in ?? ()
> (gdb) si
> [f000:e070]    0xfe070:
> ```

```
                          mov    $0xf34c2,%edx
0x0000e070 in ?? ()
(gdb) si
[f000:e076]    0xfe076:
                          jmp    0xfd15c
0x0000e076 in ?? ()
(gdb) si
[f000:d15c]    0xfd15c:
                          mov    %eax,%ecx
0x0000d15c in ?? ()
(gdb) si
[f000:d15f]    0xfd15f:
                          cli
0x0000d15f in ?? ()
(gdb) si
[f000:d160]    0xfd160:
                          cld
0x0000d160 in ?? ()
(gdb) si
[f000:d161]    0xfd161:
                          mov    $0x8f,%eax
0x0000d161 in ?? ()
(gdb) si
[f000:d167]    0xfd167:
                          out    %al,$0x70
0x0000d167 in ?? ()
(gdb) si
[f000:d169]    0xfd169:
                          in     $0x71,%al
0x0000d169 in ?? ()
(gdb) si
[f000:d16b]    0xfd16b:
                          in     $0x92,%al
0x0000d16b in ?? ()
(gdb) si
[f000:d16d]    0xfd16d:
                          or     $0x2,%al
0x0000d16d in ?? ()
(gdb) si
```

```
[f000:d16f]    0xfd16f:
                        out    %al,$0x92
0x0000d16f in ?? ()
(gdb) si
[f000:d171]    0xfd171:
                        lidtw  %cs:0x6ab8
0x0000d171 in ?? ()
(gdb) si
[f000:d177]    0xfd177:
                        lgdtw  %cs:0x6a74
0x0000d177 in ?? ()
(gdb) si
[f000:d17d]    0xfd17d:
                        mov    %cr0,%eax
0x0000d17d in ?? ()
(gdb) si
[f000:d180]    0xfd180:
                        or     $0x1,%eax
0x0000d180 in ?? ()
(gdb) si
[f000:d184]    0xfd184:
                        mov    %eax,%cr0
0x0000d184 in ?? ()
(gdb) si
[f000:d187]    0xfd187:
                        ljmpl  $0x8,$0xfd18f
0x0000d187 in ?? ()
(gdb) si
The target architecture is assumed to be i386
=> 0xfd18f:
                        mov    $0x10,%eax
0x000fd18f in ?? ()
(gdb) si
=> 0xfd194:
                        mov    %eax,%ds
0x000fd194 in ?? ()
(gdb) si
=> 0xfd196:
                        mov    %eax,%es
```

```
0x000fd196 in ?? ()
(gdb) si
=> 0xfd198:
                        mov    %eax,%ss
0x000fd198 in ?? ()
(gdb) si
=> 0xfd19a:
                        mov    %eax,%fs
0x000fd19a in ?? ()
(gdb) si
=> 0xfd19c:
                        mov    %eax,%gs
0x000fd19c in ?? ()
(gdb) si
=> 0xfd19e:
                        mov    %ecx,%eax
0x000fd19e in ?? ()
(gdb) si
=> 0xfd1a0:
                        jmp    *%edx
0x000fd1a0 in ?? ()
(gdb) si
=> 0xf34c2:
                        push   %ebx
0x000f34c2 in ?? ()
(gdb) si
=> 0xf34c3:
                        sub    $0x2c,%esp
0x000f34c3 in ?? ()
(gdb) si
=> 0xf34c6:
                        movl   $0xf5b5c,0x4(%esp)
0x000f34c6 in ?? ()
(gdb) si
=> 0xf34ce:
                        movl   $0xf447b,(%esp)
0x000f34ce in ?? ()
(gdb) si
=> 0xf34d5:
```

```
                        call   0xf099e
0x000f34d5 in ?? ()
(gdb) si
=> 0xf099e:
                        lea    0x8(%esp),%ecx
0x000f099e in ?? ()
(gdb) si
=> 0xf09a2:
                        mov    0x4(%esp),%edx
0x000f09a2 in ?? ()
(gdb) si
=> 0xf09a6:
                        mov    $0xf5b58,%eax
0x000f09a6 in ?? ()
(gdb) si
=> 0xf09ab:
                        call   0xf0574
0x000f09ab in ?? ()
(gdb) si
=> 0xf0574:
                        push   %ebp
0x000f0574 in ?? ()
(gdb) si
=> 0xf0575:
                        push   %edi
0x000f0575 in ?? ()
(gdb) si
=> 0xf0576:
                        push   %esi
0x000f0576 in ?? ()
(gdb) si
=> 0xf0577:
                        push   %ebx
0x000f0577 in ?? ()
(gdb) si
=> 0xf0578:
                        sub    $0xc,%esp
0x000f0578 in ?? ()
(gdb) si
```

```
=> 0xf057b:
                            mov    %eax,0x4(%esp)
0x000f057b in ?? ()
(gdb) si
=> 0xf057f:
                            mov    %edx,%ebp
0x000f057f in ?? ()
```

当BIOS运行时，它设置一个中断描述符表并且初始化各种设备，比如VGA显示。

初始化PCI总线和BIOS知道的所有重要的设备后，它会搜索可引导设备，例如软盘、硬盘驱动器或CD-ROM。最终，当它找到可引导磁盘，BIOS从磁盘读取引导加载程序并将控制权转移给它。

Part 2: The Boot Loader

软盘和硬盘都被分成一个个512字节的区域，被称为 *扇区* 。

扇区是磁盘最小的传输单位，每个读写操作必须是一个多多个磁盘并且关于扇区边界对齐。

如果磁盘是可引导的，第一个扇区被称为引导扇区，引导加载代码就在这个地方。当BIOS发现一个可引导的软盘或硬盘，它会将512字节的引导扇区加载到物理地址从 `0x7c00` 到 `0x7dff` ，然后用 `jmp` 指令，跳转到那里。

就像BIOS加载地址一样，这些地址是任意的，但是他们对PC来说是固定的标准化的。

由于从CD-ROM中启动的技术更新速度跟不上PC的迭代。现代的CD-ROM用2048字节的扇区而不是512字节。

在6.828中，我们将使用传统的自启动机制，意味着我们的启动加载程序必须匹配512字节。自启动加载程序包含一个汇编语言源文件 `boot/boot.s` ，和一个C语言源文件 `boot/main.c` 。

boot.S

```
#include <inc/mmu.h>

# Start the CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.

.set PROT_MODE_CSEG, 0x8         # kernel code segment selector
.set PROT_MODE_DSEG, 0x10        # kernel data segment selector
```

```
.set CR0_PE_ON,      0x1         # protected mode enable flag

.globl start
start:
  .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:               # 开启物理地址线20
  #   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       # 从端口64读
入数据存入%al
  testb   $0x2,%al                                          # %al &&
0x2
  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       # 0xdf 表示
开启A20  0xdd 表示禁用A20（对于0x60端口）
  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                 # gdtesc 指向表 gdt
  movl    %cr0, %eax              # 将%cr0寄存器的bit0置为1，进入保护模式
  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

  .code32                     # Assemble for 32-bit mode
protcseg:
  # Set up the protected-mode data segment registers
  movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS
  movw    %ax, %ss                # -> SS: Stack Segment

  # Set up the stack pointer and call into C.
  movl    $start, %esp
  call bootmain                   # -> main.c

  # If bootmain returns (it shouldn't), loop.
spin:
  jmp spin

# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
  SEG_NULL                              # null seg
  SEG(STA_X|STA_R, 0x0, 0xffffffff)     # code seg
  SEG(STA_W, 0x0, 0xffffffff)           # data seg

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

```
```

main.c

```
#include <inc/x86.h>
#include <inc/elf.h>

/**********************************************************************
*
 * This a dirt simple boot loader, whose sole job is to boot
 * an ELF kernel image from the first IDE hard disk.
 *
 * DISK LAYOUT
 *  * This program(boot.S and main.c) is the bootloader.  It should
 *    be stored in the first sector of the disk.
 *
 *  * The 2nd sector onward holds the kernel image.
 *  // 第二个扇区继续保存内核镜像

 *  * The kernel image must be in ELF format.
 *
 * BOOT UP STEPS
 *  * when the CPU boots it loads the BIOS into memory and executes it
 *
 *  * the BIOS intializes devices, sets of the interrupt routines, and
 *    reads the first sector of the boot device(e.g., hard-drive)
 *    into memory and jumps to it.
 *
 *  * Assuming this boot loader is stored in the first sector of the
 *
 *  * control starts in boot.S -- which sets up protected mode,
 *    and a stack so C code then run, then calls bootmain()
 *
 *  * bootmain() in this file takes over, reads in the kernel and jumps
to it.
 **********************************************************************
/
// The definition of struct Elf.
struct Elf {
```

```
    uint32_t e_magic;               // must equal ELF_MAGIC. 保存了 4
个 char, "\0x7FELF", 用来校验是否是一个 Elf 结构体
    uint8_t  e_elf[12];             // 应该是关于一些平台相关的设置, 关系
到如何译码和解释文件内容存  疑.
    uint16_t e_type;                // 该文件的类型
    uint16_t e_machine;             // 该文件需要的体系结构
    uint32_t e_version;             // 文件的版本
    uint32_t e_entry;               // 程序的入口地址
    uint32_t e_phoff;               // 表示 Program header table 在文
件中的偏移量(以字节计算)
    uint32_t e_shoff;               // 表示 Section header table 在文
件中的偏移量(以字节计算)
    uint32_t e_flags;               // 对 IA32 而言, 此项为 0.
    uint16_t e_ehsize;              // 表示 ELF header 大小
    uint16_t e_phentsize;           // 表示 Program header table 中每
一个条目的大小
    uint16_t e_phnum;               // 表示 Program header table 中有
多少个条目
    uint16_t e_shentsize;           // 表示 Section header table 中每
一个条目的大小
    uint16_t e_shnum;               // 表示 Section header table 中有
多少个条目
    uint16_t e_shstrndx;            // 表示包含节名称的字符串是第几个节
};

// The definition of struct Proghdr.
struct Proghdr {
  uint32_t p_type;                  // 当前 program 的段类型
  uint32_t p_offset;                // 段的第一个字节在文件中的偏移
  uint32_t p_va;                    // 段的第一个字节在文件中的虚拟地址
  uint32_t p_pa;                    // 段的第一个字节在文件中的物理地址,
在物理内存定位相关的系统中使用
  uint32_t p_filesz;                // 段在文件中的长度
  uint32_t p_memsz;                 // 段在内存中的长度
  uint32_t p_flags;                 // 与段相关的标识位
  uint32_t p_align;                 // 根据此项来确定段在文件以及内存中如何
对齐
};
```

```c
#define SECTSIZE        512
#define ELFHDR          ((struct Elf *) 0x10000) // scratch space

void readsect(void*, uint32_t);
void readseg(uint32_t, uint32_t, uint32_t);

void
bootmain(void)
{
        struct Proghdr *ph, *eph;

        // read 1st page off disk
        readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);  // SECTSIZE*8 正好4kb

        // is this a valid ELF?
        if (ELFHDR->e_magic != ELF_MAGIC)
                goto bad;

        // load each program segment (ignores ph flags)
        // 开始的位置为起始地址加上program header table在文件中的偏移量
        ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
        // 计算program header table 结束的位置
        eph = ph + ELFHDR->e_phnum;
        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);

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

bad:
        outw(0x8A00, 0x8A00);
        outw(0x8A00, 0x8E00);
        while (1)
                /* do nothing */;
}
```

```c
// Read 'count' bytes at 'offset' from kernel into physical address
'pa'.
// Might copy more than asked
void
readseg(uint32_t pa, uint32_t count, uint32_t offset)
{
	uint32_t end_pa;

	end_pa = pa + count;

	// round down to sector boundary
	// 抹掉低位的数字，锁定到扇区边界
	// 这里将512-1 再取反，然后和pa进行与运算
	pa &= ~(SECTSIZE - 1);

	// translate from bytes to sectors, and kernel starts at sector
1
	offset = (offset / SECTSIZE) + 1;

	// If this is too slow, we could read lots of sectors at a
time.
	// We'd write more to memory than asked, but it doesn't matter
--
	// we load in increasing order.
	while (pa < end_pa) {
		// Since we haven't enabled paging yet and we're using
		// an identity segment mapping (see boot.S), we can
		// use physical addresses directly.  This won't be the
		// case once JOS enables the MMU.
		// 当pa小于end_pa时，一直以扇区为单位向后读
		readsect((uint8_t*) pa, offset);
		pa += SECTSIZE;
		offset++;
	}
}

// 用来判断磁盘是否空闲
void
```

```
waitdisk(void)
{
        // wait for disk reaady
        // 当0x40这个位为1时，表示空闲
        while ((inb(0x1F7) & 0xC0) != 0x40)
                /* do nothing */;
}

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

        outb(0x1F2, 1);         // count = 1
        outb(0x1F3, offset);
        outb(0x1F4, offset >> 8);
        outb(0x1F5, offset >> 16);
        outb(0x1F6, (offset >> 24) | 0xE0);
        outb(0x1F7, 0x20);      // cmd 0x20 - read sectors

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

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

boot.asm

```
obj/boot/boot.out:     file format elf32-i386


Disassembly of section .text:

00007c00 <start>:
.set CR0_PE_ON,      0x1         # protected mode enable flag

.globl start
```

```
start:
  .code16                     # Assemble for 16-bit mode
  cli                         # Disable interrupts
    7c00:       fa                      cli
  cld                         # String operations increment
    7c01:       fc                      cld

  # Set up the important data segment registers (DS, ES, SS).
  xorw    %ax,%ax             # Segment number zero
    7c02:       31 c0                   xor    %eax,%eax
  movw    %ax,%ds             # -> Data Segment
    7c04:       8e d8                   mov    %eax,%ds
  movw    %ax,%es             # -> Extra Segment
    7c06:       8e c0                   mov    %eax,%es
  movw    %ax,%ss             # -> Stack Segment
    7c08:       8e d0                   mov    %eax,%ss

00007c0a <seta20.1>:
  # 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
    7c0a:       e4 64                   in     $0x64,%al
  testb   $0x2,%al
    7c0c:       a8 02                   test   $0x2,%al
  jnz     seta20.1
    7c0e:       75 fa                   jne    7c0a <seta20.1>

  movb    $0xd1,%al               # 0xd1 -> port 0x64
    7c10:       b0 d1                   mov    $0xd1,%al
  outb    %al,$0x64
    7c12:       e6 64                   out    %al,$0x64

00007c14 <seta20.2>:

seta20.2:
  inb     $0x64,%al               # Wait for not busy
```

```
    7c14:       e4 64                   in     $0x64,%al
  testb   $0x2,%al
    7c16:       a8 02                   test   $0x2,%al
  jnz     seta20.2
    7c18:       75 fa                   jne    7c14 <seta20.2>

  movb    $0xdf,%al               # 0xdf -> port 0x60
    7c1a:       b0 df                   mov    $0xdf,%al
  outb    %al,$0x60
    7c1c:       e6 60                   out    %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
    7c1e:       0f 01 16                lgdtl  (%esi)
    7c21:       64 7c 0f                fs jl  7c33 <protcseg+0x1>
  movl    %cr0, %eax
    7c24:       20 c0                   and    %al,%al
  orl     $CR0_PE_ON, %eax
    7c26:       66 83 c8 01             or     $0x1,%ax         # 将cr0
的第一位设置为1，进入保护模式
  movl    %eax, %cr0
    7c2a:       0f 22 c0                mov    %eax,%cr0

  # Jump to next instruction, but in 32-bit code segment.
  # Switches processor into 32-bit mode.
  ljmp    $PROT_MODE_CSEG, $protcseg
    7c2d:       ea                      .byte 0xea
    7c2e:       32 7c 08 00             xor    0x0(%eax,%ecx,1),%bh

00007c32 <protcseg>:

  .code32                     # Assemble for 32-bit mode
protcseg:
  # Set up the protected-mode data segment registers
  movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
    7c32:       66 b8 10 00             mov    $0x10,%ax
```

```
  movw    %ax, %ds                # -> DS: Data Segment
    7c36:       8e d8                   mov    %eax,%ds
  movw    %ax, %es                # -> ES: Extra Segment
    7c38:       8e c0                   mov    %eax,%es
  movw    %ax, %fs                # -> FS
    7c3a:       8e e0                   mov    %eax,%fs
  movw    %ax, %gs                # -> GS
    7c3c:       8e e8                   mov    %eax,%gs
  movw    %ax, %ss                # -> SS: Stack Segment
    7c3e:       8e d0                   mov    %eax,%ss

  # Set up the stack pointer and call into C.
  movl    $start, %esp
    7c40:       bc 00 7c 00 00          mov    $0x7c00,%esp
  call bootmain
    7c45:       e8 cb 00 00 00          call   7d15 <bootmain>

00007c4a <spin>:

  # If bootmain returns (it shouldn't), loop.
spin:
  jmp spin
    7c4a:       eb fe                   jmp    7c4a <spin>

00007c4c <gdt>:
        ...
    7c54:       ff                      (bad)
    7c55:       ff 00                   incl   (%eax)
    7c57:       00 00                   add    %al,(%eax)
    7c59:       9a cf 00 ff ff 00 00    lcall  $0x0,$0xffff00cf
    7c60:       00                      .byte 0x0
    7c61:       92                      xchg   %eax,%edx
    7c62:       cf                      iret
        ...

00007c64 <gdtdesc>:
    7c64:       17                      pop    %ss
    7c65:       00 4c 7c 00             add    %cl,0x0(%esp,%edi,2)
        ...
```

```
00007c6a <waitdisk>:
        }
}

void
waitdisk(void)
{
    7c6a:       55                      push   %ebp

static inline uint8_t
inb(int port)
{
        uint8_t data;
        asm volatile("inb %w1,%0" : "=a" (data) : "d" (port));
    7c6b:       ba f7 01 00 00          mov    $0x1f7,%edx
    7c70:       89 e5                   mov    %esp,%ebp
    7c72:       ec                      in     (%dx),%al          # 从
0x1f7端口读入数据
        // wait for disk reaady
        while ((inb(0x1F7) & 0xC0) != 0x40)
    7c73:       83 e0 c0                and    $0xffffffc0,%eax
    7c76:       3c 40                   cmp    $0x40,%al
    7c78:       75 f8                   jne    7c72 <waitdisk+0x8>
                /* do nothing */;
}
    7c7a:       5d                      pop    %ebp
    7c7b:       c3                      ret

00007c7c <readsect>:

void
readsect(void *dst, uint32_t offset)
{
    7c7c:       55                      push   %ebp
    7c7d:       89 e5                   mov    %esp,%ebp
    7c7f:       57                      push   %edi
    7c80:       8b 4d 0c                mov    0xc(%ebp),%ecx     # 偏
移量（扇区）
```

```
        // wait for disk to be ready
        waitdisk();
    7c83:       e8 e2 ff ff ff          call   7c6a <waitdisk>
}

static inline void
outb(int port, uint8_t data)
{
        asm volatile("outb %0,%w1" : : "a" (data), "d" (port));     # 
汇编指令: outb %0, %wl
                                                                    # 
eax 存储参数data, edx 存储 参数port
    7c88:       b0 01                   mov    $0x1,%al
    7c8a:       ba f2 01 00 00          mov    $0x1f2,%edx
    7c8f:       ee                      out    %al,(%dx)
    7c90:       ba f3 01 00 00          mov    $0x1f3,%edx
    7c95:       88 c8                   mov    %cl,%al
 7c97:       ee                      out    %al,(%dx)

        outb(0x1F2, 1);         // count = 1                        # 
0x1f2端口为磁盘的扇区计数
        outb(0x1F3, offset);
        outb(0x1F4, offset >> 8);
    7c98:       89 c8                   mov    %ecx,%eax
    7c9a:       ba f4 01 00 00          mov    $0x1f4,%edx
    7c9f:       c1 e8 08                shr    $0x8,%eax            # 
分别用好几个端口来存放offset的地址
    7ca2:       ee                      out    %al,(%dx)
        outb(0x1F5, offset >> 16);
    7ca3:       89 c8                   mov    %ecx,%eax
    7ca5:       ba f5 01 00 00          mov    $0x1f5,%edx
    7caa:       c1 e8 10                shr    $0x10,%eax
    7cad:       ee                      out    %al,(%dx)
        outb(0x1F6, (offset >> 24) | 0xE0);
    7cae:       89 c8                   mov    %ecx,%eax
    7cb0:       ba f6 01 00 00          mov    $0x1f6,%edx
    7cb5:       c1 e8 18                shr    $0x18,%eax
    7cb8:       83 c8 e0                or     $0xffffffe0,%eax
    7cbb:       ee                      out    %al,(%dx)
```

```
    7cbc:       b0 20                   mov    $0x20,%al
    7cbe:       ba f7 01 00 00          mov    $0x1f7,%edx            #
0x1f7端口在写操作时，写入的是命令
                                                                      #
0x20 表示读扇区
    7cc3:       ee                      out    %al,(%dx)
        outb(0x1F7, 0x20);      // cmd 0x20 - read sectors

        // wait for disk to be ready
        waitdisk();
    7cc4:       e8 a1 ff ff ff          call   7c6a <waitdisk>
        asm volatile("cld\n\trepne\n\tinsl"
    7cc9:       8b 7d 08                mov    0x8(%ebp),%edi
    7ccc:       b9 80 00 00 00          mov    $0x80,%ecx             #
循环128次
    7cd1:       ba f0 01 00 00          mov    $0x1f0,%edx
    7cd6:       fc                      cld
    7cd7:       f2 6d                   repnz insl (%dx),%es:(%edi)   #
insl 中的 l 指32位 4个字节
                                                                      #
循环128次正好是一个扇区

        // read a sector
        insl(0x1F0, dst, SECTSIZE/4);
}
    7cd9:       5f                      pop    %edi
    7cda:       5d                      pop    %ebp
    7cdb:       c3                      ret

00007cdc <readseg>:
{
    7cdc:       55                      push   %ebp
    7cdd:       89 e5                   mov    %esp,%ebp
    7cdf:       57                      push   %edi
    7ce0:       56                      push   %esi
        offset = (offset / SECTSIZE) + 1;
    7ce1:       8b 7d 10                mov    0x10(%ebp),%edi    # 第
三个参数
{
```

```
    7ce4:       53                      push   %ebx
        end_pa = pa + count;
    7ce5:       8b 75 0c                mov    0xc(%ebp),%esi     # 第二个参数
{
    7ce8:       8b 5d 08                mov    0x8(%ebp),%ebx     # 第一个参数
        offset = (offset / SECTSIZE) + 1;
    7ceb:       c1 ef 09                shr    $0x9,%edi          # 右移9位，处以512
        end_pa = pa + count;
    7cee:       01 de                   add    %ebx,%esi          # esi存储结束地址
        offset = (offset / SECTSIZE) + 1;
    7cf0:       47                      inc    %edi
 pa &= ~(SECTSIZE - 1);
    7cf1:       81 e3 00 fe ff ff       and    $0xfffffe00,%ebx
        while (pa < end_pa) {
    7cf7:       39 f3                   cmp    %esi,%ebx
    7cf9:       73 12                   jae    7d0d <readseg+0x31>
                readsect((uint8_t*) pa, offset);
    7cfb:       57                      push   %edi               # 这里与源代码的执行顺序有点区别
                                                                  # 源代码先调用readsect函数，再更新起始地址和扇区
    7cfc:       53                      push   %ebx
                offset++;
    7cfd:       47                      inc    %edi
                pa += SECTSIZE;
    7cfe:       81 c3 00 02 00 00       add    $0x200,%ebx
                readsect((uint8_t*) pa, offset);
    7d04:       e8 73 ff ff ff          call   7c7c <readsect>
                offset++;
    7d09:       58                      pop    %eax
    7d0a:       5a                      pop    %edx
    7d0b:       eb ea                   jmp    7cf7 <readseg+0x1b>
}
    7d0d:       8d 65 f4                lea    -0xc(%ebp),%esp
    7d10:       5b                      pop    %ebx
```

```
    7d11:       5e                      pop    %esi
    7d12:       5f                      pop    %edi
    7d13:       5d                      pop    %ebp
    7d14:       c3                      ret

00007d15 <bootmain>:
{
    7d15:       55                      push   %ebp
    7d16:       89 e5                   mov    %esp,%ebp
    7d18:       56                      push   %esi
    7d19:       53                      push   %ebx
        readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);
    7d1a:       6a 00                   push   $0x0
    # 从这开始push的是readseg函数的参数
    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>
        if (ELFHDR->e_magic != ELF_MAGIC)
    7d2b:       83 c4 0c                add    $0xc,%esp
    7d2e:       81 3d 00 00 01 00 7f    cmpl   $0x464c457f,0x10000
#查看0x10000附近内存可以看到

# 0x10000: 0x464c457f 0x00010101

                # 0x00000000 0x00000000

       # 0x10010: 0x00030002 0x00000001

                # 0x0010000c 0x00000034

       # 0x10020: 0x000152f8 0x00000000

                # 0x00200034 0x00280003
    7d35:       45 4c 46
    7d38:       75 37                   jne    7d71 <bootmain+0x5c>
        ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
    7d3a:       a1 1c 00 01 00          mov    0x1001c,%eax
# eax 存储ELFHDR->e_phoff
        eph = ph + ELFHDR->e_phnum;
```

```
    7d3f:       0f b7 35 2c 00 01 00    movzwl 0x1002c,%esi
# esi 存储ELFHDR->e_phnum

# 这些变量的偏移位置可从Elf结构体中推出
        ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
    7d46:       8d 98 00 00 01 00       lea    0x10000(%eax),%ebx
        eph = ph + ELFHDR->e_phnum;
    7d4c:       c1 e6 05                shl    $0x5,%esi
# 从上面的内存可以看到

# e_phentisize存放在0x1002a的位置

# 即 0x0020 32个字节 故乘32 左移5位
    7d4f:       01 de                   add    %ebx,%esi
        for (; ph < eph; ph++)
    7d51:       39 f3                   cmp    %esi,%ebx
    7d53:       73 16                   jae    7d6b <bootmain+0x56>
            readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
    7d55:       ff 73 04                pushl  0x4(%ebx)
# ebx中存放的是proghdr结构体的首地址

# 所以0x4(ebx)是段第一个字节在文件中的

# 虚拟地址
    7d58:       ff 73 14                pushl  0x14(%ebx)
# 0x14(ebx)是段在内存中的长度 p_memsz
        for (; ph < eph; ph++)
    7d5b:       83 c3 20                add    $0x20,%ebx
            readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
    7d5e:       ff 73 ec                pushl  -0x14(%ebx)
# 这里存放的是段第一个字节在文件中的

# 物理地址
    7d61:       e8 76 ff ff ff          call   7cdc <readseg>
            for (; ph < eph; ph++)
    7d66:       83 c4 0c                add    $0xc,%esp
    7d69:       eb e6                   jmp    7d51 <bootmain+0x3c>
        ((void (*)(void)) (ELFHDR->e_entry))();
    7d6b:       ff 15 18 00 01 00       call   *0x10018
```

```
}

static inline void
outw(int port, uint16_t data)
{
        asm volatile("outw %0,%w1" : : "a" (data), "d" (port));
    7d71:       ba 00 8a 00 00          mov    $0x8a00,%edx
    7d76:       b8 00 8a ff ff          mov    $0xffff8a00,%eax
    7d7b:       66 ef                   out    %ax,(%dx)
    7d7d:       b8 00 8e ff ff          mov    $0xffff8e00,%eax
    7d82:       66 ef                   out    %ax,(%dx)
    7d84:       eb fe                   jmp    7d84 <bootmain+0x6f>
```

Exercise 3.

在地址0x7c00，设置一个断点，这个位置是启动扇区将被加载的地址。比较原始的启动加载程序的源代码和 `boot.asm` 与GDB中的汇编代码。

跟踪 `boot/main.c` 中的 `bootmain()` 函数，再步入 `readsect()` 。识别出与 `readsect()` 语句对应的汇编指令。

跟踪剩下的 `readsect()` 然后返回 `bootman()` ，**识别**从磁盘读取余下内核扇区循环的 *开始* 和 *结束* 。**找出**当循环结束时，什么代码会被运行，在那里设一个断点然后继续运行到那个断点。

回答以下问题：

1. 程序从哪开始执行32位代码？什么导致了16位到32位代码的转换？

   从地址 `0x7c32` 开始， `%cr0` 设置为1导致程序进入保护模式。

2. 启动加载程序执行的 *最后* 一条指令，以及kernel刚加载完运行的 *第一条* 指令？

   最后一条指令：

   ```
   call *0x10018
   ```

   第一条指令：

   ```
   movw $0x1234, 0x472
   ```

3. kernel的第一条指令在哪?

   `0x10000c`

4. 启动加载程序时如何决定它必须读多少扇区，以便能从磁盘获取全部的kernel? 它从哪发现这个信息的?

   应该是从 `Elf` 结构体中 `e_phoff` 指向的 `Proghdr` 结构体中的 `e_shentsize` 数据项得到的。

Loading the Kernel

我们现在将看看启动加载程序中C语言部分更多的细节（`boot/main.c`）。

在这之前，是个停下来并且复习一些C语言编程基础的好时机。

Exercise 4.

- 阅读C语言中的指针编程
- 阅读5.1到5.5部分 （*The C Programming Language*）。下载pointer.c，确保你理解所有打印值从何处来。

  特别的，确保理解从行1到行6中的指针地址

  - 从行2到行4所有的值是如何到达那里的
  - 为什么行5的打印值看起来中断了

pointer.c

```
#include <stdio.h>
#include <stdlib.h>

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
    // 1: a = 0x16b806e08, b = 0x1236069c0, c = 0x104604100

    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]);
    // 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]);
    // 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]);
    // 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]);
    // 5: a[0] = 200, a[1] = 128144, a[2] = 256, a[3] = 302 在画内
存时需注意：内存为端存储，低字节放低地址

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

int
main(int ac, char **av)
{
    f();
```

```
    return 0;
}

output

1: a = 0x16b806e08, b = 0x1236069c0, c = 0x104604100
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 = 0x16b806e08, b = 0x16b806e0c, c = 0x16b806e09
```

为了理解 `boot/main.c` ，你将需要知道一个ELF二进制文件是啥。 `ELF is` **`Executable and Linkable Format`**

ELF格式全部信息

ELF简短介绍

对于6.828，你可以认为ELF可执行文件是一个携带加载信息的头文件，后面跟着几个程序段。每个程序段都是连续的代码或数据块，将被加载到特殊的内存地址。*启动加载程序不会修改代码或者数据。

一个ELF二进制文件从一个固定长度的ELF头文件开始，被一个能列出每个被加载程序段的可变长的程序头文件紧跟着。对这些ELF头文件的C语言声明在 `inc/elf.h` 。我们感兴趣的程序段是：

- `.txt` ：程序执行指令。
- `.rodata` ：只读数据，比如由C语言编译器产生的ASCII常量字符串。（但是我们不会费心设置硬件来阻止写入）
- `.data` ：这个数据段拥有程序的初始数据，比如全局变量。

当链接器计算一个程序的内存分布时，它会为没有初始化的变量保存空间，比如 `int x` 。在段中叫做 `.bss` 紧跟着 `.data` 段。

C语言要求未初始化的全局变量暂时 *“初始化”* 为0。因此，链接器只需要记录 `.bss` 段的地址和大小。然后，加载程序或者程序本身必须给 `.bss` 段**赋值为0**。

查看所有段段名称、大小、链接地址的链表，可以输入以下命令：

```
objdump -h obj/kern/kernel
//
i386-jos-elf-objdump -h obj/kern/kernel
```

大多数其他段是为了保存调试信息，一般会保存在程序的可执行文件中，但不会被程序加载程序加载到内存。

> `VMA`：又称为 *link address*。存储这个段希望从哪开始执行的内存地址。
>
> 链接器会用很多方式在二进制文件中编码链接地址，比如代码需要全局变量的地址，否则二进制文件不会工作如果它从一个没 有被链接的地址开始执行。（现代共享库广泛使用的是：生成一个位置独立的代码，不确定任何绝对地址。这需要一定的性能 和复杂度消耗，所以我们不会在6.828中使用。
>
> `LMA`：又称为 *load address*。存储这个段应该被加载到内存哪的地址。

一般情况下，链接地址和加载地址是一样的。

可以查看一下在启动加载程序中的 `.text` 段。

```
objdump -h obj/boot/boot.out
```

```
obj/boot/boot.out:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000186  00007c00  00007c00  00000074  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame     000000a8  00007d88  00007d88  000001fc  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         0000087c  00000000  00000000  000002a4  2**2
                  CONTENTS, READONLY, DEBUGGING
  3 .stabstr      00000925  00000000  00000000  00000b20  2**0
                  CONTENTS, READONLY, DEBUGGING
  4 .comment      00000029  00000000  00000000  00001445  2**0
                  CONTENTS, READONLY
```

启动加载程序用ELF *程序头* 来决定如何去加载段。这个程序头指令了ELF对象的哪些部分要加载进内存以及每个部分要占据的目标地址。你可以用下面的命令查看程序头：

```
objdump -x obj/kern/kernel
```

```
ubuntu@VM-4-14-ubuntu:~/documents/projects/mit6828/lab$ objdump -x obj/kern/kern
el

obj/kern/kernel:     file format elf32-i386
obj/kern/kernel
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c

Program Header:
    LOAD off    0x00001000 vaddr 0xf0100000 paddr 0x00100000 align 2**12
         filesz 0x0000759d memsz 0x0000759d flags r-x
    LOAD off    0x00009000 vaddr 0xf0108000 paddr 0x00108000 align 2**12
         filesz 0x0000b6a8 memsz 0x0000b6a8 flags rw-
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
         filesz 0x00000000 memsz 0x00000000 flags rwx
```

ELF对象中需要被加载进入内存的部分是那些被标记为 `LOAD` 的区域。其中 `vaddr` 表示虚拟地址， `paddr` 表示物理地址， `filesz` 和 `memsz` 表示加载区域的大小。

回到 `boot/main.c` ，每个程序头的 `ph->p_pa` 字段包含着段的目标物理地址。（在这里，它确实就是一个物理地址，尽管ELF对这个区域真正的意思的说明是模糊的。）

BIOS从地址0x7c00加载启动扇区进入内存，所以这就是内存扇区的加载地址。这也是引导扇区开始执行的地方，也是它的链接地址。

我们通过在 `boot/makefrag` 中向链接器传递 `-Ttext 0x7c00` 来设置链接地址，所以链接器在生成的代码中会产生正确的内存地址。

Exercise 5.

再次步入启动加载程序的开始一些指令，找到 *如果你get到错误的链接地址* 第一个会中断或者做一些错事的第一条指令。

然后修改 `boot/Makefrag` 中的链接地址为某个错误地址，运行命令 `make clean` ，重新用 `make` 来编译lab，再次步入启动加载程序，看看发生了什么。*不要忘记再修改回来！*

```
$(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
```

将链接地址修改为0x8c00。

```
$(OBJDIR)/boot/boot: $(BOOT_OBJS)
	@echo + ld boot/boot
	$(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x8C00 -o $@.out $^
	$(V)$(OBJDUMP) -S $@.out >$@.asm
	$(V)$(OBJCOPY) -S -O binary -j .text $@.out $@
	$(V)perl boot/sign.pl $(OBJDIR)/boot/boot
```

用 `objdump -x obj/boot/boot.out` 查看，成功修改为0x8c00.

```
ubuntu@VM-4-14-ubuntu:~/documents/projects/mit6828/lab$ objdump -h obj/boot/boot
.out

obj/boot/boot.out:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000186  00008c00  00008c00  00000074  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame     000000a8  00008d88  00008d88  000001fc  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         0000087c  00000000  00000000  000002a4  2**2
                  CONTENTS, READONLY, DEBUGGING
  3 .stabstr      00000925  00000000  00000000  00000b20  2**0
                  CONTENTS, READONLY, DEBUGGING
  4 .comment      00000029  00000000  00000000  00001445  2**0
                  CONTENTS, READONLY
```

将断点打在0x8c00，好像开始一直循环。

```
(gdb) b *0x8c00
Breakpoint 1 at 0x8c00
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
[   0:7c2d] => 0x7c2d:  ljmp   $0x8,$0x8c32
0x00007c2d in ?? ()
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
[   0:7c2d] => 0x7c2d:  ljmp   $0x8,$0x8c32
0x00007c2d in ?? ()
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
[   0:7c2d] => 0x7c2d:  ljmp   $0x8,$0x8c32
0x00007c2d in ?? ()
```

在另一个终端中，提示错误。

luzijian — ubuntu@VM-4-14-ubuntu: ~/documents/projects/mit6828/lab...

```
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.
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.
```

现在将链接地址修改回来。

查看kernel的加载地址和链接地址会发现，与启动加载程序不同的是，这两个地址不是相同的。也就是说，kernel是在告诉启动加载程序将其加载到一个低地址的内存（1 megabyte），但是它希望从高地址开始执行。我们将深入探讨我们如何在下一个段来做到这个工作。

ELF头中的区域，除了段信息之外还有一个区域对我们来说很重要，叫做 `e_entry` 。这个区域保存着 *entry point* 的链接地址：程序应该执行的地址，也就是程序 `.text` 段的内存地址。你可以用下面的命令查看entry point：

```
objdump -f obj/kern/kernel
```

```
ubuntu@VM-4-14-ubuntu:~/documents/projects/mit6828/lab$ objdump -f obj/kern/kern
el

obj/kern/kernel:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c
```

你现在应该能理解 `boot/main.c` 中最小的ELF加载器。它从磁盘中读取kernel的每个段放入内存中段的加载地址，然后跳转到kernel的entry point。

Exercise 6.

We can examine memory using GDB's x command. The GDB manual has full details, but for now, it is enough to know that the command x/*N* x *ADDR* prints *N* words of memory at *ADDR* . (Note that both ' x 's in the command are lowercase.) *Warning* : The size of a word is not a universal standard. In GNU assembly, a word is two bytes (the 'w' in xorw, which stands for word, means 2 bytes).

Reset the machine (exit QEMU/GDB and start them again). Examine the 8 words of memory at 0x00100000 at the point the BIOS enters the boot loader, and then again at the point the boot loader enters the kernel. Why are they different? What is there at the second breakpoint? (You do not really need to use QEMU to answer this question. Just think.)

当BIOS进入启动加载程序时

```
(gdb) x/8xw 0x00100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
```

当启动加载程序进入kernel时

```
(gdb) x/8xw 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x2000b812      0x220f0011      0xc0200fd8
```

Part 3: The Kernel

现在我们将开始检查这个小型的JOS内核更细节一点了（你最后将会写一些代码）。就像启动加载程序，内核从一些汇编代码开始，这些汇编代码会设置一些东西，所以C语言代码能够正确执行。

Using virtual memory to work around position dependence

当你查看启动加载程序的链接地址和加载地址，你会发现它们完美重合，但是对于*kernel* 的链接地址和加载地址来说，它们有很大的差异性。

操作系统内核通常喜欢被链接和运行在一个很高的虚拟地址，比如 `0xf0100000` ，为了将进程较低的虚拟地址留给用户程序使用。在下一个lab中，这个安排的原因会更清楚。

很多机器在地址0xf0100000没有物理地址，所以我们不能指望将内核存储在那里。相反，我们将使用 *进程的内存管理硬件* 来映射虚地址0xf0100000（这个地址是内核代码希望运行的开始地址）为物理地址0x00100000（这个地址是启动加载程序加载内核的物理地址）。尽管内核的虚拟地址已经足够高来留下很多的地址空间给用户进程，他还是将被加载到物理内存1MB的地方，就在

BIOS的上方。

事实上，在下一个实验，我们将映射全部PC物理地址空间的底部256MB，从物理地址0x00000000到0x0fffffff，到虚拟地址0xf0000000到0xffffffff。

现在，我们将仅仅映射前4MB到物理内存。我们用 `kern/entrypgdir.c` 中手写，静态初始化的页目录和页表来映射。直到 `kern/entry.S` 设置 `CR0_PG` 标志位，内存引用都被认为是物理地址。一旦 `CR0_PG` 标志位被设置，内存引用就是虚拟地址，它被虚拟内存硬件映射为物理地址。 `entry_pgdir` 翻译虚拟地址从0xf0000000到0xf0400000为物理地址0x00000000到0x00400000，也将虚拟地址0x00000000到0x00400000映射到物理地址0x00000000到0x00400000。

任何不在这两个地址范围内的虚拟地址都会导致硬件异常，因为我们还没有设置中断处理，这种异常会导致QEMU宕机并退出（或者无休止的重启，如果你没有使用QEMU的补丁版本）。

Exercise 7.

Use QEMU and GDB to trace into the JOS kernel and stop at the `movl %eax, %cr0` . Examine memory at 0x00100000 and at 0xf0100000. Now, single step over that instruction using the stepi GDB command. Again, examine memory at 0x00100000 and at 0xf0100000. Make sure you understand what just happened.

What is the first instruction *after* the new mapping is established that would fail to work properly if the mapping weren't in place? Comment out（注释）the `movl %eax, %cr0` in `kern/entry.S` , trace into it, and see if you were right.

```
(gdb) x/8xw 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x2000b812      0x220f0011      0xc0200fd8
(gdb) x/8xw 0xf0100000
0xf0100000 <_start+4026531828>: 0x00000000      0x00000000      0x00000000      0
x00000000
0xf0100010 <entry+4>:   0x00000000      0x00000000      0x00000000      0x000000
00
(gdb) si
=> 0x100025:    mov    %eax,%cr0
0x00100025 in ?? ()
(gdb) si
=> 0x100028:    mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) x/8xw 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x2000b812      0x220f0011      0xc0200fd8
(gdb) x/8xw 0xf0100000
0xf0100000 <_start+4026531828>: 0x1badb002      0x00000000      0xe4524ffe      0
x7205c766
0xf0100010 <entry+4>:   0x34000004      0x2000b812      0x220f0011      0xc0200f
d8
```

bit	label	descrption
0	Pe	Protected mode enable
1	Mp	Monitor co–processor
2	Em	Emulation
3	Ts	Task switched
4	Et	extension type
5	Ne	Numeric error
16	Wp	write protect
18	Am	alignment mask
29	Nw	Not–write through

30	Cd	cache disable
31	Pg	paging

上表是 `%cr0` 寄存器每个位的作用。

```
mov %eax, %cr0
```

是为了打开保护模式。

Formatted Printing to the Console

大多数人认为使用 `printf()` 这样的函数是理所当然的，有时甚至认为它们是C语言的原始函数。但是在一个OS的内核中，我们不得不自己完成所有的I/O操作。

通读 `kern/printf.c` , `lib/printfmt.c` 和 `kern/console.c` ，确保你理解它们之间的关系。在之后的lab中， `printfmt.c` 为什么位于分开的 `lib` 目录将会变得明确。

`kern/printf.c`

```
/ Simple implementation of cprintf console output for the kernel,
// based on printfmt() and the kernel console's cputchar().

#include <inc/types.h>
#include <inc/stdio.h>
#include <inc/stdarg.h>


static void
putch(int ch, int *cnt)                          // 将字符ch放入输出
缓冲区
{
        cputchar(ch);
```

```
        *cnt++;
}
int
vcprintf(const char *fmt, va_list ap)
{
        int cnt = 0;

        vprintfmt((void*)putch, &cnt, fmt, ap);       // vcprintf 调用下
面的vprintfmt
        return cnt;
}
int
cprintf(const char *fmt, ...)
{
        va_list ap;
        int cnt;

        va_start(ap, fmt);                            // 初始化ap指针指向
参数列表
        cnt = vcprintf(fmt, ap);                      // 这里开始调用
vcprintf
        va_end(ap);                                   // 将ap指针释放并设
置为NULL

        return cnt;
}
```

`lib/printfmt.c`

```
// Stripped-down primitive printf-style formatting routines,
// used in common by printf, sprintf, fprintf, etc.
// This code is also used by both the kernel and user programs.

#include <inc/types.h>
#include <inc/stdio.h>
#include <inc/string.h>
#include <inc/stdarg.h>
#include <inc/error.h>

/*
```

```
 * Space or zero padding and a field width are supported for the
numeric
 * formats only.
 *
 * The special format %e takes an integer error code
 * and prints a string describing the error.
 * The integer may be positive or negative,
 * so that -E_NO_MEM and E_NO_MEM are equivalent.
 */

static const char * const error_string[MAXERROR] =
{
	[E_UNSPECIFIED] = "unspecified error",
	[E_BAD_ENV]     = "bad environment",
	[E_INVAL]       = "invalid parameter",
	[E_NO_MEM]      = "out of memory",
	[E_NO_FREE_ENV] = "out of environments",
	[E_FAULT]       = "segmentation fault",
};

/*
 * Print a number (base <= 16) in reverse order,
 * using specified putch function and associated pointer putdat.
 */
static void
printnum(void (*putch)(int, void*), void *putdat,
	 unsigned long long num, unsigned base, int width, int padc)
{
	// first recursively(递归地) print all preceding (more
significant) digits
	if (num >= base) {
		printnum(putch, putdat, num / base, base, width - 1,
padc);
	} else {
		// print any needed pad characters before first digit
		while (--width > 0)
			putch(padc, putdat);
	}
```

```
        // then print this (the least significant) digit
        putch("0123456789abcdef"[num % base], putdat);
}

// Get an unsigned int of various possible sizes from a varargs list,
// depending on the lflag parameter.    从参数列表获取一个多种可能大小的无符号整数
static unsigned long long
getuint(va_list *ap, int lflag)
{
        if (lflag >= 2)
                return va_arg(*ap, unsigned long long);             //从ap指向的参数列表中取出一个ull类型的值
        else if (lflag)
                return va_arg(*ap, unsigned long);
        else
                return va_arg(*ap, unsigned int);
}

// Same as getuint but signed - can't use getuint
// because of sign extension
static long long
getint(va_list *ap, int lflag)
{
        if (lflag >= 2)
                return va_arg(*ap, long long);
        else if (lflag)
                return va_arg(*ap, long);
        else
                return va_arg(*ap, int);
}


// Main function to format and print a string.
void printfmt(void (*putch)(int, void*), void *putdat, const char *fmt, ...);

void
```

```
vprintfmt(void (*putch)(int, void*), void *putdat, const char *fmt,
va_list ap)
// 这里的fmt格式化字符可参考链接 C语言格式化字符
https://blog.csdn.net/MyLinChi/article/details/53116760
// 这里的第一个参数可以理解为*回调函数*
{
        register const char *p;
        register int ch, err;
        unsigned long long num;
        int base, lflag, width, precision, altflag;
        char padc;

        while (1) {
                while ((ch = *(unsigned char *) fmt++) != '%') {     //
判断参数fmt是不是字符'%'
                        if (ch == '\0')                              //
'\0' 是字符串结束标志
                                return;
                        putch(ch, putdat);                           //
putch是一个参数函数，暂时不知道是干嘛的
                                                                     //
putch是将参数ch放入输出缓冲区并输出
                                                                     //
putdat用于记录输出的字符数目
                }

                // Process a %-escape sequence                       //
开始处理%后面的字符
                padc = ' ';
                width = -1;
                precision = -1;
                lflag = 0;
                altflag = 0;
        reswitch:
                switch (ch = *(unsigned char *) fmt++) {

                // flag to pad on the right
                case '-':                                            //
'-'表示左对齐
```

```
                padc = '-';
                goto reswitch;

            // flag to pad with 0's instead of spaces
            case '0':                                                    //
对所有数字格式用前导0填充字段宽度
                padc = '0';
                goto reswitch;

            // width field
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
                for (precision = 0; ; ++fmt) {                           //
计算数字的精度
                    precision = precision * 10 + ch - '0';
                    ch = *fmt;
                    if (ch < '0' || ch > '9')
                        break;
                }
                goto process_precision;

            case '*':
                precision = va_arg(ap, int);                             //
精度存储在ap所指的参数列表
                goto process_precision;

            case '.':
                if (width < 0)
                    width = 0;
                goto reswitch;
```

```
		case '#':                                   // 对'o'类，输出时加'o'；对'x'类，输出时加'0x'
			altflag = 1;
			goto reswitch;

		process_precision:
			if (width < 0)
				width = precision, precision = -1;
			goto reswitch;

		// long flag (doubled for long long)
		case 'l':
			lflag++;
			goto reswitch;

		// character
		case 'c':
			putch(va_arg(ap, int), putdat);             // 直接输出
			break;

		// error message
		case 'e':
			err = va_arg(ap, int);
			if (err < 0)
				err = -err;
			if (err >= MAXERROR || (p = error_string[err]) == NULL)
				printfmt(putch, putdat, "error %d", err);
			else
				printfmt(putch, putdat, "%s", p);
			break;

		// string
		case 's':
			if ((p = va_arg(ap, char *)) == NULL)
				p = "(null)";
			if (width > 0 && padc != '-')
```

```
                                for (width -= strnlen(p, precision);
width > 0; width--)
                                        putch(padc, putdat);
                        for (; (ch = *p++) != '\0' && (precision < 0 ||
--precision >= 0); width--)
                                if (altflag && (ch < ' ' || ch > '~'))
                                        putch('?', putdat);
                                else
                                        putch(ch, putdat);
                        for (; width > 0; width--)
                                putch(' ', putdat);
                        break;

                // (signed) decimal
                case 'd':
                        num = getint(&ap, lflag);
                        if ((long long) num < 0) {
                                putch('-', putdat);
                                num = -(long long) num;
                        }
                        base = 10;
                        goto number;

                // unsigned decimal
                case 'u':
                        num = getuint(&ap, lflag);
                        base = 10;
                        goto number;

                // (unsigned) octal
                case 'o':
                        // Replace this with your code.
                        num = getuint(&ap, lflag);
                        base = 8;
                        goto number;
                        /*
                        putch('X', putdat);
                        putch('X', putdat);
                        putch('X', putdat);
```

```
                        break;
                        */

                // pointer
                case 'p':
// 打印指针
                        putch('0', putdat);
                        putch('x', putdat);
                        num = (unsigned long long)
                                (uintptr_t) va_arg(ap, void *);
                        base = 16;
                        goto number;

                // (unsigned) hexadecimal
                case 'x':
                        num = getuint(&ap, lflag);
                        base = 16;
                number:
                        printnum(putch, putdat, num, base, width,
padc);      // 打印出不同进制的数字
                        break;

                // escaped '%' character
// 转义'%'字符

// 如果格式化字符串为'%%', 则输出'%'
                case '%':
                        putch(ch, putdat);
                        break;

                // unrecognized escape sequence - just print it
literally     // 直接输出未识别的转义序列
                default:
                        putch('%', putdat);
                        for (fmt--; fmt[-1] ≠ '%'; fmt--)
                                /* do nothing */;
                        break;
                }
        }
```

```c
}

void
printfmt(void (*putch)(int, void*), void *putdat, const char *fmt, ...)
{
	va_list ap;

	va_start(ap, fmt);
	vprintfmt(putch, putdat, fmt, ap);
	va_end(ap);
}

struct sprintbuf {
	char *buf;
	char *ebuf;
	int cnt;
};

static void
sprintputch(int ch, struct sprintbuf *b)
{
	b->cnt++;
	if (b->buf < b->ebuf)
		*b->buf++ = ch;
}

int
vsnprintf(char *buf, int n, const char *fmt, va_list ap)
{
	struct sprintbuf b = {buf, buf+n-1, 0};

	if (buf == NULL || n < 1)
		return -E_INVAL;

	// print the string to the buffer
	vprintfmt((void*)sprintputch, &b, fmt, ap);

	// null terminate the buffer
	*b.buf = '\0';
```

```
        return b.cnt;
}

int
snprintf(char *buf, int n, const char *fmt, ...)                // 将
字符串写入buf
{
        va_list ap;
        int rc;

        va_start(ap, fmt);
        rc = vsnprintf(buf, n, fmt, ap);
        va_end(ap);

        return rc;
}
```

`kern/console.c`

```
/* See COPYRIGHT for copyright information. */

#include <inc/x86.h>
#include <inc/memlayout.h>
#include <inc/kbdreg.h>
#include <inc/string.h>
#include <inc/assert.h>

#include <kern/console.h>

static void cons_intr(int (*proc)(void));
static void cons_putc(int c);

// Stupid I/O delay routine necessitated by historical PC design flaws
static void
delay(void)
{
        inb(0x84);
        inb(0x84);
```

```
        inb(0x84);
        inb(0x84);
}

/***** Serial I/O code *****/
// 串口地址   x86的I/O编址是独立编址      参考地址:
https://bochs.sourceforge.io/techspec/PORTS.LST

//https://www.twblogs.net/a/5b89e8d02b71775d1ce46b55
#define COM1            0x3F8   /*w   serial port, transmitter holding
register, which contains the
                                      character to be sent. Bit 0 is
sent first.
                                      bit 7-0   data bits when DLAB=0
(Divisor Latch Access Bit)
                                 *r   receiver buffer register, which
contains the received character
                                      Bit 0 is received first
                                      bit 7-0   data bits when DLAB=0
(Divisor Latch Access Bit)
                                 *r/w divisor latch low byte when
DLAB=1
                                 */

#define COM_RX          0       // In:  Receive buffer (DLAB=0)
#define COM_TX          0       // Out: Transmit buffer (DLAB=0)
#define COM_DLL         0       // Out: Divisor Latch Low (DLAB=1)
// DLL和DLM合起来用于配制分频器
#define COM_DLM         1       // Out: Divisor Latch High (DLAB=1)
#define COM_IER         1       // Out: Interrupt Enable Register
#define   COM_IER_RDI   0x01    //   Enable receiver data interrupt
#define COM_IIR         2       // In:  Interrupt ID Register
#define COM_FCR         2       // Out: FIFO Control Register
#define COM_LCR         3       // Out: Line Control Register
#define   COM_LCR_DLAB  0x80    //   Divisor latch access bit
#define   COM_LCR_WLEN8 0x03    //   Wordlength: 8 bits
#define COM_MCR         4       // Out: Modem Control Register
#define   COM_MCR_RTS   0x02    // RTS complement
#define   COM_MCR_DTR   0x01    // DTR complement
```

```
#define   COM_MCR_OUT2  0x08    // Out2 complement
#define COM_LSR         5       // In:  Line Status Register
#define   COM_LSR_DATA  0x01    //   Data available
#define   COM_LSR_TXRDY 0x20    //   Transmit buffer avail
#define   COM_LSR_TSRE  0x40    //   Transmitter off

static bool serial_exists;

static int
serial_proc_data(void)
{
        if (!(inb(COM1+COM_LSR) & COM_LSR_DATA))        // COM1+COM_LSR
端口号变为0x3FD，该端口的最低bit位显示数据
                                                        // 是否就绪
                return -1;
        return inb(COM1+COM_RX);
}

void
serial_intr(void)
{
        if (serial_exists)
                cons_intr(serial_proc_data);
}

static void
serial_putc(int c)
{
        int i;

        for (i = 0;
             !(inb(COM1 + COM_LSR) & COM_LSR_TXRDY) && i < 12800;
                                                        // 判断0x3FD端口
的传输buffer是否就绪
                                                        // 12800/512=25
             i++)
                delay();
```

```
        outb(COM1 + COM_TX, c);                          // 将c内容写入
0x3F8端口
}

static void
serial_init(void)
{
        // Turn off the FIFO
        outb(COM1+COM_FCR, 0);

        // Set speed; requires DLAB latch
        outb(COM1+COM_LCR, COM_LCR_DLAB);                // COM_LCR对于
out来说是线控制寄存器
                                                         // COM_LCR_DLAB
是除数锁访问位
        outb(COM1+COM_DLL, (uint8_t) (115200 / 9600));   // DLL和DLM合起
来配制分频器
        outb(COM1+COM_DLM, 0);

        // 8 data bits, 1 stop bit, parity off; turn off DLAB latch
        outb(COM1+COM_LCR, COM_LCR_WLEN8 & ~COM_LCR_DLAB);
                                                         //

        // No modem controls
        outb(COM1+COM_MCR, 0);                           // MCR为调制解调
器用于控制Modem
                                                         // Modem有调制和
调解的作用
                                                         // 调制：数字信
号—>模拟信号 调解：模拟信号—>数字信号
        // Enable rcv interrupts
        outb(COM1+COM_IER, COM_IER_RDI);                 // RDI: receive
data interrupt

        // Clear any preexisting overrun indications and interrupts
        // 清除任何提前存在的超时的迹象和中断
        // Serial port doesn't exist if COM_LSR returns 0xFF
        serial_exists = (inb(COM1+COM_LSR) != 0xFF);
```

```
        (void) inb(COM1+COM_IIR);                       // interrupt id
register
        (void) inb(COM1+COM_RX);

}



/***** Parallel port output code *****/
// For information on PC parallel port programming, see the class
References
// page.

static void
lpt_putc(int c)
{
        int i;
        // 0x378端口是并行端口 0x379是状态端口 0x37A是控制端口
        for (i = 0; !(inb(0x378+1) & 0x80) && i < 12800; i++)
                delay();
        outb(0x378+0, c);                              // 0x378端口是数据
        outb(0x378+2, 0x08|0x04|0x01);                 // bit0=1:strobe
bit2=0:initialize printer
                                                       // bit3=1:select
printer
        outb(0x378+2, 0x08);                           // select printer
猜测是将c输出
}



/***** Text-mode CGA/VGA display output *****/

static unsigned addr_6845;                             // 控制台的输出地址
static uint16_t *crt_buf;                              // 控制台的输出内容
static uint16_t crt_pos;                               // 光标位置，输出缓
冲字符的个数
```

```c
static void
cga_init(void)
{
	volatile uint16_t *cp;
	uint16_t was;
	unsigned pos;

	cp = (uint16_t*) (KERNBASE + CGA_BUF);
	was = *cp;
	*cp = (uint16_t) 0xA55A;
	if (*cp != 0xA55A) {
		cp = (uint16_t*) (KERNBASE + MONO_BUF);
		addr_6845 = MONO_BASE;
	} else {
		*cp = was;
		addr_6845 = CGA_BASE;
	}

	/* Extract cursor location */
	// 获得光标位置
	outb(addr_6845, 14);
	pos = inb(addr_6845 + 1) << 8;
	outb(addr_6845, 15);
	pos |= inb(addr_6845 + 1);

	crt_buf = (uint16_t*) cp;
	crt_pos = pos;
}



static void
cga_putc(int c)
{
	// if no attribute given, then use black on white
	if (!(c & ~0xFF))
	// 高两个字节改变输出背景颜色 低两个字节为数据内容
	// preference:https://blog.csdn.net/cy295957410/article/details/108436730
```

```
		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) {
	  // 光标位置超过屏幕大小
		int i;
	  // memmove(1,2,3) 从2中拷贝3个字节进入1
		memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE -
CRT_COLS) * sizeof(uint16_t));
	  // 把所有行向上移动一行
		for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
			crt_buf[i] = 0x0700 | ' ';
		crt_pos -= CRT_COLS;
```

```
        }

        /* move that little blinky thing */
        // 移动光标
        // 存放光标位置的寄存器编号是14、15
        // 高八位输入到14号寄存器
        // 低八位输入到15号寄存器
        outb(addr_6845, 14);
        // 先向0x3D4写入寄存器编号 再通过0x3D5写读写寄存器
        //
preferen:https://blog.csdn.net/cy295957410/article/details/108436730
        outb(addr_6845 + 1, crt_pos >> 8);
        outb(addr_6845, 15);
        outb(addr_6845 + 1, crt_pos);
}


/***** Keyboard input code *****/

#define NO              0

#define SHIFT           (1<<0)
#define CTL             (1<<1)
#define ALT             (1<<2)

#define CAPSLOCK        (1<<3)
#define NUMLOCK         (1<<4)
#define SCROLLLOCK      (1<<5)

#define E0ESC           (1<<6)

static uint8_t shiftcode[256] =
{
        [0x1D] = CTL,
        [0x2A] = SHIFT,
        [0x36] = SHIFT,
        [0x38] = ALT,
        [0x9D] = CTL,
        [0xB8] = ALT
```

```c
};

static uint8_t togglecode[256] =
{
        [0x3A] = CAPSLOCK,
        [0x45] = NUMLOCK,
        [0x46] = SCROLLLOCK
};

static uint8_t normalmap[256] =
{
        NO,   0x1B, '1',  '2',  '3',  '4',  '5',  '6',  // 0x00
        '7',  '8',  '9',  '0',  '-',  '=',  '\b', '\t',
        'q',  'w',  'e',  'r',  't',  'y',  'u',  'i',  // 0x10
        'o',  'p',  '[',  ']',  '\n', NO,   'a',  's',
        'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';',  // 0x20
        '\'', '`',  NO,   '\\', 'z',  'x',  'c',  'v',
        'b',  'n',  'm',  ',',  '.',  '/',  NO,   '*',  // 0x30
        NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
        NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
        '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
        '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
        [0xC7] = KEY_HOME,            [0x9C] = '\n' /*KP_Enter*/,
        [0xB5] = '/' /*KP_Div*/,      [0xC8] = KEY_UP,
        [0xC9] = KEY_PGUP,            [0xCB] = KEY_LF,
        [0xCD] = KEY_RT,              [0xCF] = KEY_END,
        [0xD0] = KEY_DN,              [0xD1] = KEY_PGDN,
        [0xD2] = KEY_INS,             [0xD3] = KEY_DEL
};

static uint8_t shiftmap[256] =
{
        NO,   033,  '!',  '@',  '#',  '$',  '%',  '^',  // 0x00
        '&',  '*',  '(',  ')',  '_',  '+',  '\b', '\t',
        'Q',  'W',  'E',  'R',  'T',  'Y',  'U',  'I',  // 0x10
        'O',  'P',  '{',  '}',  '\n', NO,   'A',  'S',
        'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  // 0x20
        '"',  '~',  NO,   '|',  'Z',  'X',  'C',  'V',
        'B',  'N',  'M',  '<',  '>',  '?',  NO,   '*',  // 0x30
```

```c
        NO,   ' ',  NO,   NO,   NO,   NO,   NO,   NO,
        NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',  // 0x40
        '8',  '9',  '-',  '4',  '5',  '6',  '+',  '1',
        '2',  '3',  '0',  '.',  NO,   NO,   NO,   NO,   // 0x50
        [0xC7] = KEY_HOME,            [0x9C] = '\n' /*KP_Enter*/,
        [0xB5] = '/' /*KP_Div*/,      [0xC8] = KEY_UP,
        [0xC9] = KEY_PGUP,            [0xCB] = KEY_LF,
        [0xCD] = KEY_RT,              [0xCF] = KEY_END,
        [0xD0] = KEY_DN,              [0xD1] = KEY_PGDN,
        [0xD2] = KEY_INS,             [0xD3] = KEY_DEL
};

#define C(x) (x - '@')

static uint8_t ctlmap[256] =
{
        NO,      NO,      NO,      NO,      NO,      NO,      NO,
 NO,
        NO,      NO,      NO,      NO,      NO,      NO,      NO,
 NO,
        C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),
 C('I'),
        C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),
 C('S'),
        C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),
 NO,
        NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),
 C('V'),
        C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,
 NO,
        [0x97] = KEY_HOME,
        [0xB5] = C('/'),                [0xC8] = KEY_UP,
        [0xC9] = KEY_PGUP,              [0xCB] = KEY_LF,
        [0xCD] = KEY_RT,                [0xCF] = KEY_END,
        [0xD0] = KEY_DN,                [0xD1] = KEY_PGDN,
        [0xD2] = KEY_INS,               [0xD3] = KEY_DEL
};

static uint8_t *charcode[4] = {
```

```
        normalmap,
        shiftmap,
        ctlmap,
        ctlmap
};

/*
 * Get data from the keyboard.  If we finish a character, return it.
Else 0.
 * Return -1 if no data.
 */
static int
kbd_proc_data(void)
// 键盘寄存器有4个8bit的寄存器
// 状态寄存器和控制寄存器两者共用一个端口0x64 输入缓冲区和输出缓冲区共用一个端口
0x60
{
        int c;
        uint8_t stat, data;
        static uint32_t shift;

        // 得到键盘控制器状态
        // preference:https://juejin.cn/post/7002181336048484383
        // preference:https://zhuanlan.zhihu.com/p/402293362
        stat = inb(KBSTATP);
        if ((stat & KBS_DIB) == 0)
                return -1;
        // Ignore data from mouse.
        if (stat & KBS_TERR)
                return -1;

        // 读数据  端口60
        data = inb(KBDATAP);

        // 通码最高位第7位（从第0位开始）为1。 断码最高位第7位为0
        // 通码：键被按下时的编码 断码：键弹起时的编码
        if (data == 0xE0) {
                // E0 escape character
                // E0 表示多个字节
```

```
		shift |= E0ESC;
		return 0;
	} else if (data & 0x80) {
		// Key released
		data = (shift & E0ESC ? data : data & 0x7F);
		// 若断码为控制键，将shift中的记录信息清楚
		shift &= ~(shiftcode[data] | E0ESC);
		return 0;
	} else if (shift & E0ESC) {
		// Last character was an E0 escape; or with 0x80
		data |= 0x80;
		shift &= ~E0ESC;
	}

	shift |= shiftcode[data];
	shift ^= togglecode[data];

	c = charcode[shift & (CTL | SHIFT)][data];
	if (shift & CAPSLOCK) {
		if ('a' <= c && c <= 'z')
			c += 'A' - 'a';
		else if ('A' <= c && c <= 'Z')
			c += 'a' - 'A';
	}

	// Process special keys
	// Ctrl-Alt-Del: reboot
	if (!(~shift & (CTL | ALT)) && c == KEY_DEL) {
		cprintf("Rebooting!\n");
		outb(0x92, 0x3); // courtesy of Chris Frost
	}

	return c;
}

void
kbd_intr(void)
{
	cons_intr(kbd_proc_data);
```

```
}

static void
kbd_init(void)
{
}



/***** General device-independent console code *****/
// Here we manage the console input buffer,
// where we stash characters received from the keyboard or serial port
// whenever the corresponding interrupt occurs.

#define CONSBUFSIZE 512

static struct {
	uint8_t buf[CONSBUFSIZE];
	uint32_t rpos;
	uint32_t wpos;
} cons;

// called by device interrupt routines to feed input characters
// into the circular console input buffer.
static void
cons_intr(int (*proc)(void))
{
	int c;

	while ((c = (*proc)()) != -1) {
		if (c == 0)
			continue;
		cons.buf[cons.wpos++] = c;
		if (cons.wpos == CONSBUFSIZE)
			cons.wpos = 0;
	}
}
```

```c
// return the next input character from the console, or 0 if none
waiting
int
cons_getc(void)
{
	int c;

	// poll for any pending input characters,
	// so that this function works even when interrupts are
disabled
	// (e.g., when called from the kernel monitor).
	// 将COM端口读入的数据放入输入端口 过程中有回调函数
	serial_intr();
	// 将键盘中的输入放入缓冲区
	kbd_intr();

	// grab the next character from the input buffer.
	// 从缓冲区读数据
	if (cons.rpos != cons.wpos) {
		c = cons.buf[cons.rpos++];
		if (cons.rpos == CONSBUFSIZE)
			cons.rpos = 0;
		return c;
	 // 返回控制台的输入
	}
	return 0;
}

// output a character to the console
static void
cons_putc(int c)
{
	serial_putc(c);
	lpt_putc(c);
	cga_putc(c);
}

// initialize the console devices
void
```

```c
cons_init(void)
{
	cga_init();
	kbd_init();
	serial_init();

	if (!serial_exists)
		cprintf("Serial port does not exist!\n");
}


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

void
cputchar(int c)
{
	cons_putc(c);
}

int
getchar(void)
{
	int c;

	while ((c = cons_getc()) == 0)
		/* do nothing */;
	return c;
}

int
iscons(int fdnum)
{
	// used by readline
	return 1;
```

Exercise 8.

We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment.

```
 // It is in the function vprintfmt()
             case 'o':
                     // Replace this with your code.
                    num = getuint(&ap, lflag);
                    base = 8;
                    goto number;
```

能够回答下面的问题：

1. 解释 `printf.c` 和 `console.c` 之间的接口。特别的，`console.c` 导出的是什么函数？这个函数是如何被 `print.c` 使用的？

   接口是 `cputchar()` 函数。

   console.c

   ```
   void
   cputchar(int c)
   {
           cons_putc(c);
   }
   ```

   printf.c

   ```
   static void
   putch(int ch, int *cnt)                                 // 将字符ch放入输出缓冲区
   {
           cputchar(ch);
           *cnt++;
   }
   ```

2. 解释下面 `console.c` 中的代码：

```
// 如果光标位置超出屏幕大小
    if (crt_pos >= CRT_SIZE) {
        int i;
// 将所有行向上移动一行
        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;
    }
```

3. Trace the execution of the following code step-by-step:

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

- In the call to `cprintf()`, to what does `fmt` point? To what does `ap` point?

  `fmt` 指向前面的字符串

  `ap` 指向后面的x, y, z变量

- List (in order of execution) each call to `cons_putc`, `va_arg`, and `vcprintf`. For `cons_putc`, list its argument as well.

  For `va_arg`, list what `ap` points to before and after the call. For `vcprintf` list the values of its two arguments.

  - For `cons_putc`: `cgi_putc()`, `cputchar()`

    For `va_arg`: `getuint()`, `getint()`, `vprintfmt()`, `pitch()`

    For `vcprintf`: `cprintf()`

  - `cons_putc()` 的参数是 `int` 整形。

    `va_arg()` 的 `ap` 指针在调用完指向的地址是调用前地址加上第二个参数的大小。

    `vcprintf()` 的参数是 `const char* fmt`, `va_list ap`

4. Run the following code.

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

What is the output? Explain how this output is arrived at in the step-by-step manner of the previous exercise. Here's an ASCII table that maps bytes to characters.

输出是：He110 World

格式化字符串%x 会将数字以16进制的方式打印出来

格式化字符串%s 会将内容以字符串的形式输出，每次读两个字节，且从低到高读出的是：`0x72`，`0x6c`，`0x64`，对应的字符为 `r`，`l`，`d`。

The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set `i` to in order to yield the same output? Would you need to change `57616` to a different value?

5. In the following code, what is going to be printed after `'y='`? (note: the answer is not a specific value.) Why does this happen?

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

   格式化字符串漏洞

6. Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change `cprintf` or its interface so that it would still be possible to pass it a variable number of arguments?

   需要改变 `ap` 指针的移动方向

The stack

在这个lab最后的练习中，我们将探索更多细节关于C语言在x86架构上使用栈的方式，并且在进程写一个能够打印栈的 `backtrace` 的有用的新内核函数。这个 `backtrace` 是一个列表，存放着导致当前执行点的嵌套调用指令的指令指针的值。

Exercise 9.

Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

1. 通过 `relocated()` 函数
2. 通过 `sub $0x.., %esp` 来预留栈空间

栈的 ebp 和 esp 机制是很有用的。举个例子：当一个特定的函数导致了一个 `assert` 错误或者 `panic`，因为错误的参数被传入，但是你并不知道是谁传入了这个错误的参数。这时，栈的 `backtrace` 就可以帮你找到那个函数。

Exercise 10.

To become familiar with the C calling conventions on the x86, find the address of the `test_backtrace` function in `obj/kern/kernel.asm`, set a breakpoint there, and examine what happens each time it gets called after the kernel starts. How many 32-bit words does each recursive nesting level of `test_backtrace` push on the stack, and what are those words?

Note that, for this exercise to work properly, you should be using the patched version of QEMU available on the tools page or on Athena. Otherwise, you'll have to manually translate all breakpoint and memory addresses to linear addresses.

- 找到 `test_backtrace` 的地址 **0xf0100040**

  `mon_backtrace()` 地址 **0xf0100883**

  `cprintf()` 地址**0xf0100af8**

```
f0100040 <test_backtrace>:
#include <kern/console.h>

// Test the stack backtrace function (lab 1 only)
void
test_backtrace(int x)
{
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 f8 06 ff ff       lea
-0xf908(%ebx),%eax
f010005d:       50                      push   %eax
f010005e:       e8 e6 09 00 00          call   f0100a49
<cprintf>
        if (x > 0)
f0100063:       83 c4 10                add    $0x10,%esp
f0100066:       85 f6                   test   %esi,%esi
f0100068:       7f 2b                   jg     f0100095
<test_backtrace+0x55>
                test_backtrace(x-1);
        else
                mon_backtrace(0, 0, 0);
f010006a:       83 ec 04                sub    $0x4,%esp
f010006d:       6a 00                   push   $0x0
f010006f:       6a 00                   push   $0x0
f0100071:       6a 00                   push   $0x0
f0100073:       e8 0b 08 00 00          call   f0100883
<mon_backtrace>
f0100078:       83 c4 10                add    $0x10,%esp
        cprintf("leaving test_backtrace %d\n", x);
f010007b:       83 ec 08                sub    $0x8,%esp
f010007e:       56                      push   %esi
f010007f:       8d 83 14 07 ff ff       lea
-0xf8ec(%ebx),%eax
f0100085:       50                      push   %eax
f0100086:       e8 be 09 00 00          call   f0100a49
<cprintf>
}
```

```
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
                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>
```

查看 `test_backtrace()` 的C语言代码。

```
// 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);
}
```

以及 `mon_backtrace()` 的源码。

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
        // Your code here.
        return 0;
}
```

- How many 32-bit words does each recursive nesting level of `test_backtrace` push on the stack, and what are those words?

  `ebp` , `esi` , `ebx`

上面的练习应该给你了你需要去完成一个栈 `backtrace` 函数的信息，也就是调用 `mon_backtrace()` 函数。这个函数的原型已将放在 `kern/monitor.c` 中。你可以完全使用C语言完成，但是你可能会发现 `inc/x86.h` 中的 `read_ebp()` 函数有用。

你还必须将这个新函数挂接到内核监视器的命令列表中，以便能被用户交互调用。

这个 `backtrace` 函数应该展示一个如下格式的函数调用帧列表：

```
Stack backtrace:
  ebp f0109e58  eip f0100a62  args 00000001 f0109e80 f0109e98 f0100ed2
00000031
  ebp f0109ed8  eip f01000d6  args 00000000 00000000 f0100058 f0109f28
00000061
  ...
```

每一行都包含 `ebp` , `eip` , `args` . `ebp` 的值指向那个函数所使用栈的基指针：比如，就在函数进入之后的栈指针位置，然后函数的序言代码设置基指针。列表中的 `eip` 值是函数的返回指令指针。当函数返回时，指令地址将控制返回到的地方。返回指令指针一般指向 `call` 指令之后的指令。最后， `args` 后的五个十六进制是问题里函数前五个参数，这些参数就在函数被调用之前被存放在栈中。如果函数不需要五个参数，当然，并不是这五个值都是有用的。 *（为什么不能回溯代码来检测实际上有多少参数呢？如何解决这个限制？）*

第一行打印的反映了当前正在执行的函数，叫做 `mon_backtrace` ，第二行反映一个调用 `mon_backtrace` 的函数，第三行反映一个调用刚刚那个函数的函数……

你应该打印出所有突出的栈帧。通过学习 `kern/entry.s` 你将发现有一个简单的方式去发现什么时候停下来。

这里有一些你在阅读 *K&R Chapter 5* 时值得记住的点，在后面的练习和将来的labs中会有用。

- 如果 `int *p = (int*)100` ， `(int)p + 1` 和 `(int)(p+1)` 是不一样的数字：第一个是101，而第二个是104。当一个指针加上一个整数，在第二个例子中，这个整数会被隐式地乘上这个对象指针的size大小。
- `p[i]` 和 `*(p+i)` 是一样的，指向 `p` 指针指向的内存中第 `i` 个对象。上面的增加规则帮助了定义工作当对象类型大小大于一个字节。
- `&p[i]` 和 `(p+i)` 是相同的，两个都表示 `p` 指针所指内存第 `i` 个对象的地址。

尽管大多数的C程序从不需要在指针和整形之间进行转换，操作系统却经常这样做。无论什么时候当你看见一个增加操作涉及内存地址时，问问你自己这是整形增加还是指针增加并且确认被加的值是否被合适地倍乘。

Exercise 11.

Implement the backtrace function as specified above. Use the same format as in the example, since otherwise the grading script will be confused. When you think you have it working right, run make grade to see if its output conforms to what our grading script expects, and fix it if it doesn't. *After* you have handed in your Lab 1 code, you are welcome to change the output format of the backtrace function any way you like.

If you use `read_ebp()` , note that GCC may generate "optimized" code that calls `read_ebp()` *before* `mon_backtrace()` 's function prologue, which results in an incomplete stack trace (the stack frame of the most recent function call is missing). While we have tried to disable optimizations that cause this reordering, you may want to examine the assembly of `mon_backtrace()` and make sure the call to `read_ebp()` is happening after the function prologue.

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
	// Your code here.
	cprintf("Stack backtrace:\n");

	// get the current ebp
uint32_t ebp;
	asm volatile("movl %%ebp, %0" : "=r" (ebp));

	// get all the six ebp(s) from the current ebp through the
stack mechanism
uint32_t* (ebpp[6]);
	ebpp[0] = (uint32_t*) ebp;
	for(int i = 1; i <= 5; ++i){
		ebpp[i] = (uint32_t*) *ebpp[i-1];
	}
```

```c
	// print eip, args of the each ebp
	for(int i = 0; i <= 5; ++i){
		cprintf("  %d ebp %08x eip %08x args %08x %08x %08x %08x %08x\n",
			i, ebpp[i], *(ebpp[i] + 1), *(ebpp[i] + 2), *(ebpp[i] + 3), *(ebpp[i] + 4), *(ebpp[i] + 5),
				*(ebpp[i] + 6));
	}
	return 0;
```

Result

```
Stack backtrace:

0 ebp f010ff18 eip f0100078 args 00000000 00000000 00000000 f010004a f0111308

1 ebp f010ff38 eip f01000a1 args 00000000 00000001 f010ff78 f010004a f0111308

2 ebp f010ff58 eip f01000a1 args 00000001 00000002 f010ff98 f010004a f0111308

3 ebp f010ff78 eip f01000a1 args 00000002 00000003 f010ffb8 f010004a f0111308

4 ebp f010ff98 eip f01000a1 args 00000003 00000004 00000000 f010004a f0111308

5 ebp f010ffb8 eip f01000a1 args 00000004 00000005 00000000 f010004a f0111308
```

这时，你的 `backtrace` 函数应该告诉了你使得 `mon_backtrace()` 被执行的函数调用者地址。然而，在实战中，你通常想知道那些地址对应的函数名称。例如，你可能想知道是哪个函数会包含导致你的内核崩溃的BUG。

为了帮助你完成这个功能，我们提供了一个 `debuginfo_eip()` 函数，这个函数寻找在标志表中的 `eip` 并返回那个地址的调试信息。这个函数被定义于 `kern/kdebug.c` 。

Exercise 12.

Modify your stack backtrace function to display, for each `eip`, the function name, source file name, and line number corresponding to that `eip`.

In `debuginfo_eip`, where do `__STAB_*` come from? This question has a long answer; to help you to discover the answer, here are some things you might want to do:

- look in the file `kern/kernel.ld` for `__STAB_*`

```
/* Simple linker script for the JOS kernel.
   See the GNU ld 'info' manual ("info ld") to learn the
syntax. */

OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)

SECTIONS
{
        /* Link the kernel at this address: "." means the
current address */
        . = 0xF0100000;

        /* AT(...) gives the load address of this section,
which tells
           the boot loader where to load the kernel in physical
memory */
        /* 当标志是在段内声明，那么它是相对于这个段的基址
           其它地方声明的则是绝对符号（绝对地址） */
        .text : AT(0x100000) {
                *(.text .stub .text.* .gnu.linkonce.t.*)
        }
        /* PROVIDE keyword 可以定义一个标志，当这个标志仅被引用但没有
被定义时 */
        PROVIDE(etext = .);     /* Define the 'etext' symbol to
this value */

        .rodata : {
                *(.rodata .rodata.* .gnu.linkonce.r.*)
```

```
        }

        /* Include debugging information in kernel memory */
        .stab : {
                PROVIDE(__STAB_BEGIN__ = .);
                *(.stab);
                PROVIDE(__STAB_END__ = .);
                BYTE(0)         /* Force the linker to allocate
    space
                                   for this section */
        }

        .stabstr : {
                PROVIDE(__STABSTR_BEGIN__ = .);
                *(.stabstr);
                PROVIDE(__STABSTR_END__ = .);
                BYTE(0)         /* Force the linker to allocate
    space
                                   for this section */
        }

        /* Adjust the address for the data segment to the next
    page */
        . = ALIGN(0x1000);

        /* The data segment */
        .data : {
                *(.data)
        }

        .bss : {
                PROVIDE(edata = .);
                *(.bss)
                PROVIDE(end = .);
                BYTE(0)
        }
```

```
      /DISCARD/ : {
              *(.eh_frame .note.GNU-stack)
      }

 }
```

- run objdump -h obj/kern/kernel

  ```c
  obj/kern/kernel:     file format elf32-i386

  Sections:
  Idx Name          Size      VMA       LMA       File off  Algn
    0 .text         00001a59  f0100000  00100000  00001000  2**4
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
    1 .rodata       00000708  f0101a60  00101a60  00002a60  2**5
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
    2 .stab         00003c55  f0102168  00102168  00003168  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
    3 .stabstr      0000196f  f0105dbd  00105dbd  00006dbd  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 00000044  f0113000  00113000  00014000
2**2
                    CONTENTS, ALLOC, LOAD, DATA
    9 .bss          00000648  f0113060  00113060  00014060  2**5
                    CONTENTS, ALLOC, LOAD, DATA
   10 .comment      00000029  00000000  00000000  000146a8  2**0
                    CONTENTS, READONLY
```

- run objdump -G obj/kern/kernel

  the content is too much

- run gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -DJOS_KERNEL -gstabs -c -S kern/init.c, and look at init.s.

```
        .file   "init.c"
        .stabs  "kern/init.c",100,0,2,.Ltext0
        .text
.Ltext0:
        .stabs  "gcc2_compiled.",60,0,0,0
        .stabs
 "int:t(0,1)=r(0,1);-2147483648;2147483647;",128,0,0,0
        .stabs  "char:t(0,2)=r(0,2);0;127;",128,0,0,0
        .stabs  "long
int:t(0,3)=r(0,3);-0;4294967295;",128,0,0,0
        .stabs  "unsigned
int:t(0,4)=r(0,4);0;4294967295;",128,0,0,0
        .stabs  "long unsigned
int:t(0,5)=r(0,5);0;-1;",128,0,0,0
        .stabs  "__int128:t(0,6)=r(0,6);0;-1;",128,0,0,0
        .stabs  "__int128
unsigned:t(0,7)=r(0,7);0;-1;",128,0,0,0
        .stabs  "long long
int:t(0,8)=r(0,8);-0;4294967295;",128,0,0,0
        .stabs  "long long unsigned
int:t(0,9)=r(0,9);0;-1;",128,0,0,0
        .stabs  "short
int:t(0,10)=r(0,10);-32768;32767;",128,0,0,0
        .stabs  "short unsigned
int:t(0,11)=r(0,11);0;65535;",128,0,0,0
        .stabs  "signed
char:t(0,12)=r(0,12);-128;127;",128,0,0,0
        .stabs  "unsigned
char:t(0,13)=r(0,13);0;255;",128,0,0,0
        .stabs  "float:t(0,14)=r(0,1);4;0;",128,0,0,0
        .stabs  "double:t(0,15)=r(0,1);8;0;",128,0,0,0
```

```
	.stabs	"long double:t(0,16)=r(0,1);16;0;",128,0,0,0
	.stabs	"_Float32:t(0,17)=r(0,1);4;0;",128,0,0,0
	.stabs	"_Float64:t(0,18)=r(0,1);8;0;",128,0,0,0
	.stabs	"_Float128:t(0,19)=r(0,1);16;0;",128,0,0,0
	.stabs	"_Float32x:t(0,20)=r(0,1);8;0;",128,0,0,0
	.stabs	"_Float64x:t(0,21)=r(0,1);16;0;",128,0,0,0
	.stabs	"_Decimal32:t(0,22)=r(0,1);4;0;",128,0,0,0
	.stabs	"_Decimal64:t(0,23)=r(0,1);8;0;",128,0,0,0
	.stabs	"_Decimal128:t(0,24)=r(0,1);16;0;",128,0,0,0
	.stabs	"void:t(0,25)=(0,25)",128,0,0,0
	.stabs	"./inc/stdio.h",130,0,0,0
	.stabs	"./inc/stdarg.h",130,0,0,0
	.stabs	"va_list:t(2,1)=(2,2)=
(2,3)=ar(2,4)=r(2,4);0;-1;;0;0;
(2,5)=xs__va_list_tag:",128,0,0,0
	.stabn	162,0,0,0
	.stabn	162,0,0,0
	.stabs	"./inc/string.h",130,0,0,0
	.stabs	"./inc/types.h",130,0,0,0
	.stabs	"bool:t(4,1)=(4,2)=eFalse:0,True:1,;",128,0,0,0
	.stabs	" :T(4,3)=efalse:0,true:1,;",128,0,0,0
	.stabs	"int8_t:t(4,4)=(0,12)",128,0,0,0
	.stabs	"uint8_t:t(4,5)=(0,13)",128,0,0,0
	.stabs	"int16_t:t(4,6)=(0,10)",128,0,0,0
	.stabs	"uint16_t:t(4,7)=(0,11)",128,0,0,0
	.stabs	"int32_t:t(4,8)=(0,1)",128,0,0,0
	.stabs	"uint32_t:t(4,9)=(0,4)",128,0,0,0
	.stabs	"int64_t:t(4,10)=(0,8)",128,0,0,0
	.stabs	"uint64_t:t(4,11)=(0,9)",128,0,0,0
	.stabs	"intptr_t:t(4,12)=(4,8)",128,0,0,0
	.stabs	"uintptr_t:t(4,13)=(4,9)",128,0,0,0
	.stabs	"physaddr_t:t(4,14)=(4,9)",128,0,0,0
	.stabs	"ppn_t:t(4,15)=(4,9)",128,0,0,0
	.stabs	"size_t:t(4,16)=(4,9)",128,0,0,0
	.stabs	"ssize_t:t(4,17)=(4,8)",128,0,0,0
	.stabs	"off_t:t(4,18)=(4,8)",128,0,0,0
	.stabn	162,0,0,0
	.stabn	162,0,0,0
	.section	.rodata.str1.1,"aMS",@progbits,1
```

```
.LC0:
        .string "entering test_backtrace %d\n"
.LC1:
        .string "leaving test_backtrace %d\n"
        .text
        .p2align 4,,15
        .stabs  "test_backtrace:F(0,25)",36,0,0,test_backtrace
        .stabs  "x:P(0,1)",64,0,0,3
        .globl  test_backtrace
        .type   test_backtrace, @function
test_backtrace:
        .stabn  68,0,13,.LM0-.LFBB1
.LM0:
.LFBB1:
.LFB0:
        .cfi_startproc
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        .stabn  68,0,14,.LM1-.LFBB1
.LM1:
        movl    %edi, %esi
        .stabn  68,0,13,.LM2-.LFBB1
.LM2:
        movl    %edi, %ebx
        .stabn  68,0,14,.LM3-.LFBB1
.LM3:
        leaq    .LC0(%rip), %rdi
        xorl    %eax, %eax
        call    cprintf@PLT
        .stabn  68,0,15,.LM4-.LFBB1
.LM4:
        testl   %ebx, %ebx
        jg      .L6
        .stabn  68,0,18,.LM5-.LFBB1
.LM5:
        xorl    %edx, %edx
        xorl    %esi, %esi
        xorl    %edi, %edi
```

```
        call    mon_backtrace@PLT
.L3:
        .stabn  68,0,19,.LM6-.LFBB1
.LM6:
        movl    %ebx, %esi
        leaq    .LC1(%rip), %rdi
        xorl    %eax, %eax
        .stabn  68,0,20,.LM7-.LFBB1
.LM7:
        popq    %rbx
        .cfi_remember_state
        .cfi_def_cfa_offset 8
        .stabn  68,0,19,.LM8-.LFBB1
.LM8:
        jmp     cprintf@PLT
        .p2align 4,,10
        .p2align 3
.L6:
        .cfi_restore_state
        .stabn  68,0,16,.LM9-.LFBB1
.LM9:
        leal    -1(%rbx), %edi
        call    test_backtrace
        jmp     .L3
        .cfi_endproc
.LFE0:
        .size   test_backtrace, .-test_backtrace
.Lscope1:
        .section        .rodata.str1.1
.LC2:
        .string "6828 decimal is %o octal!\n"
        .text
        .p2align 4,,15
        .stabs  "i386_init:F(0,25)",36,0,0,i386_init
        .globl  i386_init
        .type   i386_init, @function
i386_init:
        .stabn  68,0,24,.LM10-.LFBB2
.LM10:
```

```
.LFBB2:
.LFB1:
        .cfi_startproc
        .stabn  68,0,30,.LM11-.LFBB2
.LM11:
        leaq    edata(%rip), %rdi
        leaq    end(%rip), %rdx
        .stabn  68,0,24,.LM12-.LFBB2
.LM12:
        subq    $8, %rsp
        .cfi_def_cfa_offset 16
        .stabn  68,0,30,.LM13-.LFBB2
.LM13:
        xorl    %esi, %esi
        subq    %rdi, %rdx
        call    memset@PLT
        .stabn  68,0,34,.LM14-.LFBB2
.LM14:
        call    cons_init@PLT
        .stabn  68,0,36,.LM15-.LFBB2
.LM15:
        leaq    .LC2(%rip), %rdi
        movl    $6828, %esi
        xorl    %eax, %eax
        call    cprintf@PLT
        .stabn  68,0,39,.LM16-.LFBB2
.LM16:
        movl    $5, %edi
        call    test_backtrace
        .p2align 4,,10
        .p2align 3
.L8:
        .stabn  68,0,43,.LM17-.LFBB2
.LM17:
        xorl    %edi, %edi
        call    monitor@PLT
        jmp     .L8
        .cfi_endproc
.LFE1:
```

```
        .size   i386_init, .-i386_init
.Lscope2:
        .section        .rodata.str1.1
.LC3:
        .string "kernel panic at %s:%d: "
.LC4:
        .string "\n"
        .text
        .p2align 4,,15
        .stabs  "_panic:F(0,25)",36,0,0,_panic
        .stabs  "file:P(0,26)=*(0,2)",64,0,0,5
        .stabs  "line:P(0,1)",64,0,0,4
        .stabs  "fmt:P(0,26)",64,0,0,3
        .globl  _panic
        .type   _panic, @function
_panic:
        .stabn  68,0,59,.LM18-.LFBB3
.LM18:
.LFBB3:
.LFB2:
        .cfi_startproc
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        subq    $208, %rsp
        .cfi_def_cfa_offset 224
        testb   %al, %al
        movq    %rcx, 56(%rsp)
        movq    %r8, 64(%rsp)
        movq    %r9, 72(%rsp)
        je      .L11
        movaps  %xmm0, 80(%rsp)
        movaps  %xmm1, 96(%rsp)
        movaps  %xmm2, 112(%rsp)
        movaps  %xmm3, 128(%rsp)
        movaps  %xmm4, 144(%rsp)
        movaps  %xmm5, 160(%rsp)
        movaps  %xmm6, 176(%rsp)
```

```
        movaps  %xmm7, 192(%rsp)
.L11:
        .stabn  68,0,59,.LM19-.LFBB3
.LM19:
        movq    %fs:40, %rax
```

Stab format

```
#define N_GSYM          0x20    // global symbol
#define N_FNAME         0x22    // F77 function name
#define N_FUN           0x24    // procedure name
#define N_STSYM         0x26    // data segment variable
#define N_LCSYM         0x28    // bss segment variable
#define N_MAIN          0x2a    // main function name
#define N_PC            0x30    // global Pascal symbol
#define N_RSYM          0x40    // register variable
#define N_SLINE         0x44    // text segment line number
#define N_DSLINE        0x46    // data segment line number
#define N_BSLINE        0x48    // bss segment line number
#define N_SSYM          0x60    // structure/union element
#define N_SO            0x64    // main source file name
#define N_LSYM          0x80    // stack variable
#define N_BINCL         0x82    // include file beginning
#define N_SOL           0x84    // included source file name
#define N_PSYM          0xa0    // parameter variable
#define N_EINCL         0xa2    // include file end
#define N_ENTRY         0xa4    // alternate entry point
#define N_LBRAC         0xc0    // left bracket
#define N_EXCL          0xc2    // deleted include file
#define N_RBRAC         0xe0    // right bracket
#define N_BCOMM         0xe2    // begin common
#define N_ECOMM         0xe4    // end common
#define N_ECOML         0xe8    // end common (local name)
#define N_LENG          0xfe    // length of preceding entry

// 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
};
```

`stab_binsearch()`

```
static void
stab_binsearch(const struct Stab *stabs, int *region_left, int
*region_right,
	       int type, uintptr_t addr)
{
	int l = *region_left, r = *region_right, any_matches =
0;

	while (l <= r) {
		int true_m = (l + r) / 2, m = true_m;

		// search for earliest stab with right type
		while (m >= l && stabs[m].n_type != type)
			m--;
		if (m < l) {	// no match in [l, m]
			l = true_m + 1;
			continue;
		}

		// actual binary search
		any_matches = 1;
		if (stabs[m].n_value < addr) {
			*region_left = m;
			l = true_m + 1;
		} else if (stabs[m].n_value > addr) {
			*region_right = m - 1;
			r = m - 1;
		} else {
			// exact match for 'addr', but continue
loop to find
			// *region_right
```

```
                        *region_left = m;
                        l = m;
                        addr++;
                }
        }

        if (!any_matches)
                *region_right = *region_left - 1;
        else {
                // find rightmost region containing 'addr'
                for (l = *region_right;
                     l > *region_left && stabs[l].n_type !=
    type;

                     l--)
                        /* do nothing */;
                *region_left = l;
        }
    }
```

- see if the bootloader loads the symbol table in memory as part of loading the kernel binary

Complete the implementation of `debuginfo_eip` by inserting the call to `stab_binsearch` to find the line number for an address.

Add a `backtrace` command to the kernel monitor, and extend your implementation of `mon_backtrace` to call `debuginfo_eip` and print a line for each stack frame of the form:

```
K> backtrace
Stack backtrace:
  ebp f010ff78  eip f01008ae  args 00000001 f010ff8c 00000000
f0110580 00000000
         kern/monitor.c:143: monitor+106
  ebp f010ffd8  eip f0100193  args 00000000 00001aac 00000660
00000000 00000000
         kern/init.c:49: i386_init+59
  ebp f010fff8  eip f010003d  args 00000000 00000000 0000ffff
10cf9a00 0000ffff
         kern/entry.S:70: <unknown>+0
K>
```

Each line gives the file name and line within that file of the stack frame's `eip`, followed by the name of the function and the offset of the `eip` from the first instruction of the function (e.g., `monitor+106` means the return `eip` is 106 bytes past the beginning of `monitor`).

Be sure to print the file and function names on a separate line, to avoid confusing the grading script.

Tip: printf format strings provide an easy, albeit obscure, way to print non-null-terminated strings like those in STABS tables. `printf("%.*s", length, string)` prints at most `length` characters of `string`. Take a look at the printf man page to find out why this works.

You may find that some functions are missing from the backtrace. For example, you will probably see a call to `monitor()` but not to `runcmd()`. This is because the compiler in-lines some function calls. Other optimizations may cause you to see unexpected line numbers. If you get rid of the `-O2` from `GNUMakefile`, the backtraces may make more sense (but your kernel will run more slowly).

code segment

```
// 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[lline].n_desc;
        }else{
                return -1;
        }
```

result

```
6828 decimal is XXX 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:
  0 ebp f010ff18 eip f0100078 args 00000000 00000000 00000000
f010004a f0111308
       kern/init.c:18:test_backtrace+56
  1 ebp f010ff38 eip f01000a1 args 00000000 00000001 f010ff78
f010004a f0111308
       kern/init.c:16:test_backtrace+97
  2 ebp f010ff58 eip f01000a1 args 00000001 00000002 f010ff98
f010004a f0111308
       kern/init.c:16:test_backtrace+97
  3 ebp f010ff78 eip f01000a1 args 00000002 00000003 f010ffb8
f010004a f0111308
       kern/init.c:16:test_backtrace+97
```

```
  4 ebp f010ff98 eip f01000a1 args 00000003 00000004 00000000
f010004a f0111308
       kern/init.c:16:test_backtrace+97
  5 ebp f010ffb8 eip f01000a1 args 00000004 00000005 00000000
f010004a f0111308
       kern/init.c:16:test_backtrace+97
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!
```