Lab 1: Booting a PC

第一部分 PC引导

第一部分着重于熟悉以下三个问题：

- x86汇编语言
- QEMU x86模拟器
- PC启动引导过程

环境搭建好以后，开始依次学习各个问题。

1.x86汇编语言

第一个练习的目的是介绍x86汇编语言和PC引导进程,然后开始熟悉QEMU和QEMU调试。这部分暂不需编写代码，但是大致上还是要有一个基本的理解，并为后续问题做准备。

通过阅读 PC Assembly Language 对x86汇编语言完成快速、基本全面的了解。大致把握NASM（Intel语法）和GNU/GAS（AT&T语法）的基本区别，基本语法的变化和转换已在Brennan's Guide to Inline Assembly中给有详细的说明。

> **练习一：**
>
> 熟悉网站the 6.828 reference page上已有的汇编语言资料，无需详细阅读，但要有一个基本的认识以便后续阅读编写代码时检索。阅读Brennan's Guide to Inline Assembly的Sytax部分，包含两种语法的差异和转换的概括

x86汇编语言编程的最终参考还是在于Intel的指令集结构参考，在the 6.828 reference page上可以看到有两种相关的不同风格：

1. 一种是HTML版的旧版本，见80386 Programmer's Reference Manual，这一版简短易查阅，而且概括了我们可能用到的所有x86汇编语言的特性
2. 另一种则是最完整、最新版本、最强大的Intel的用户手册，见IA-32 Intel Architecture Software Developer's Manuals，概括了我们可能用得到的和将来可能用到的。另外有一版比较类似的手册集available from AMD。

将上述手册事先保存下来以便后续查阅。

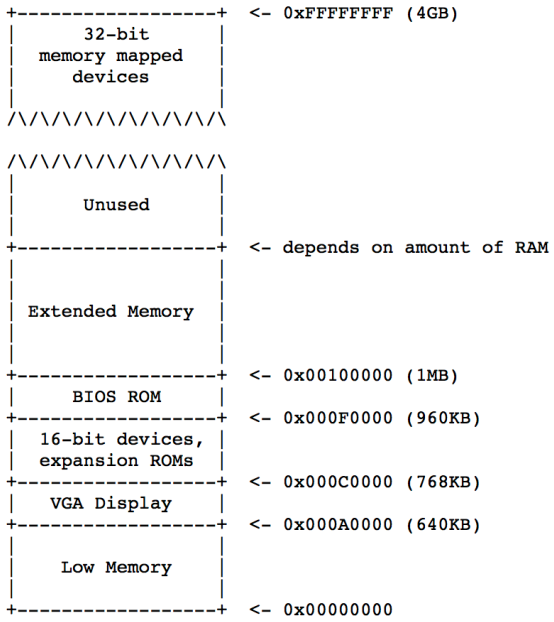
2.模拟x86

我们将使用模拟器来模拟x86，很大程度上简化了调试。在6.828中我们使用QEMU模拟器，其本身仅提供非常有限的调试支持，“ 但QEMU可以作为GNU调试器（GDB）的远程调试目标，我们将在本实验中使用它来逐步完成早期启动过程（来自Google翻译）”。

首先将Lab 1的文件克隆到目录下，然后在lab目录下键入 `make` 以搭建微型6.828引导加载程序和内核，然后便可以键入 `make qemu` 运行QEMU。

3.PC物理地址空间

简单深入了解一下PC启动的的细节，一个硬连的PC物理地址空间一般有如下层次：



早期的PC基于16位Intel 8088处理器只有1MB的物理地址空间，所以寻址范围在0x00000000到0x000FFFFF之间，其中前640KB的“Low memory”部分是唯一可用的RAM。

从640KB处到1MB处之间的384KB的部分则被硬件维护用于特殊地方，例如视频显示缓冲区和保存于非易失性存储器的固件，这里边最重要的部分是占据了最后64KB的基本输入输出系统（BIOS）。早期PC中BIOS一般保存在ROM中，但当前的PC则保存在可更新闪存中。BIOS负责系统初始化，例如激活视频卡、检查安装内存量。在初始化完成之后BIOS则从软盘、硬盘、CD-ROM或网络中加载OS，然后便将控制权交与OS手中。

Intel最终通过80286和80386处理器攻破1MB的围墙时，物理内存空间增大到16MB乃至4GB，但是PC架构师仍然保留了原始的低1MB的内存布局以确保现有软件的向后兼容性。现代PC的内存中因此出现了从0x000A0000到0x00100000的缺口，这个缺口将内存分为了低内存或称常规内存的部分（最开始的640KB）和扩展内存（其余的全部内存）。另外，在32位物理地址空间最顶部、于所有RAM上边的部分内存，现在通常由BIOS保存用于32位PCI设备。

最近的x86处理器可以支持空间超过4GB的物理RAM，在这种情况下BIOS必须在32位可寻址区域顶部的系统RAM中留存另一个缺口，为这些32位设备预留映射空间。因为设计的局限，JOS仅使用PC的前256MB物理内存，所以目前我们暂时假设所有的PC都只有一个32位的物理地址空间。

4.ROM BIOS

这部分将用QEMU调试工具观察IA32兼容计算机的引导方式。一个终端执行 `make qemu-gdb` ，另一个执行 `make gdb` 。结果中有一行 `[f000:fff0] 0xffff0: ljmp $0xf000,$0xe05b` 说明：

1. PC从物理地址0x000ffff0开始执行，就在ROM BIOS内存区域（64KB）的最顶端
2. PC由CS=0xf000和IP=0xfff0开始执行
3. 第一条讲要执行的指令为jmp指令，转移到到CS=0xf000和IP=0xe05b

解释：PC中的BIOS采用硬连逻辑，物理地址范围为0x000F0000到0x000FFFFF，上述设计确保了BIOS总可以在机器首次启动或者系统重启后率先得到控制权——这是至关重要的，因为启动时RAM中不存在处理器需要执行的软件。QEMU仿真器带有BIOS，BIOS位于处理器模拟物理地址空间的对应位置。在处理器复位时，模拟处理器进入实模式，将CS设置为0xf000、IP设置为0xfff0，以便从CS: IP段地址开始执行。

那么段地址0xf000: 0xfff0是如何转化为物理地址的呢？公式：*物理地址=16 * 段 + 偏移量*。所以
*16 * 0xf000 + 0xfff0*
= 0xf0000 + 0xfff0
= 0xffff0
0xffff0位于BIOS尾部前16 Byte，所以BIOS先要执行jmp指令转移到BIOS区域的靠前的位置。

> **练习二：**
>
> 执行GDB的步进 `si` 命令来跟踪ROM BIOS的几个命令，并猜测其作用。无需究其细节，掌握大致意思即可。可以查阅Phil Storrs I/O Ports Description和6.828 reference materials page中的资料。

BIOS在运行时创建中断描述符表、初始化各种设备比如如VGA显示。初始化PCI总线和所有重要的设备之后，BIOS将会搜索可启动设备，如软盘、硬盘驱动器或CD-ROM，最终当其找到可引导硬盘时，BIOS会从磁盘读取引导加载程序并将控制权转移给它。

第二部分 引导加载程序

软盘、硬盘一般被分为以512B为一个单位的扇区。一个扇区是一个盘的最小传输粒度，每次读 / 写操作都必须是针对一个或若干个扇区，并且在山区边界上对齐。一个可启动的盘的第一个扇区通常称作引导扇区，存放引导加载程序的代码段。当BIOS搜索到可启动的的软盘或者硬盘时就会将521B的引导扇区加载到内存空间的0x7c00到0x7dff区域，然后使用一个跳转指令将CS: IP设置为0000: 7c00从二将控制权传递给引导加载程序。和BIOS加载地址一样，这些地址的确定是相当随意的，但对于PC是固定化、标准化的。

从CD-ROM中引导的技术在PC变革时代中出现得比较晚，其扇区大小一般不是512B，但是对于6.828而言我们仍然用的是传统的硬盘驱动引导机制，也就是说引导加载程序必须安排在512B的空间中。

引导加载程序包含了一个汇编语言源文件 `boot/boot.s` 和一个C源文件 `boot/main.c` ，仔细阅读这些源文件，确保搞懂这些文件在做什么。一个引导加载程序必须有一下两个主要功能：

1. 引导加载程序必须能够将处理器从实模式转换为32位保护模式，只有在这个模式下软件才可以访问到物理地址空间中1MB以上的部分。介绍保护模式的相关资料很多，但是目前只需要知道在保护模式下从段地址到物理地址的转换和实模式不同，并且在转换后偏移量是32位而不是16位。
2. 引导加载程序必须能够通过x86的特殊IO指令直接访问IDE磁盘设备寄存器，从硬盘读取内核。

了解了引导加载程序的源代码以后，可以读一下文件 `obj/boot/boot.asm` ，这个文件是GNU的makefile在编译引导加载程序之后创建的反汇编。这个反汇编文件帮助我们看到所有的引导加载程序存放在物理内存的准确位置，而且可以通过这个文件追踪在GDB中通过引导加载程序时发生的情况。类似地， `obj/kern/kernel` 包含了JOS内核的反汇编，在调试的时候往往有所指导。

用命令 `b` 可以在GDB中设置断点，例如 `b *0x7c00` 就可以在0x7c00处设置断点。然后可以通过命令 `c` 或 `si` 继续执行。还可以通过 `x/i` 命令查看内存中的指令，内存中第一条指令会自动打印，这条命令的完整语法是 `x/Ni ADDRESS` ， `N` 表示将要连续反汇编的指令条数， `ADDRESS` 则表示内存中开始反汇编的起始地址。

> **练习三：**
>
> 阅读 lab tools guide，尤其是GDB命令的部分。在0x7c00地址（引导扇区加载的地址）处设置断点，然后执行程序直到到达这个断点，同时跟踪 `boot/boot.s` 中的代码，用源代码和反汇编文件 `obj/boot/boot.asm` 来追踪你的位置。并且在GDB中用 `x/i` 命令以对引导加载程序中的指令序列进行反汇编，然后对比原始引导加载程序源代码和 `boot.asm` 以及GDB中的反汇编。

回答以下问题：

- 处理器从哪里开始执行32位模式？究竟是什么导致从16位到32位的转换？
- 执行引导加载程序的最后一条指令是什么？它刚加载的内核的第一条指令是什么？
- 内核的第一条指令在哪里？
- 引导加载程序是如何确定需要读多少个扇区才能从磁盘中获取整个内核？它是从哪里获取到这个信息的？

加载内核

现在深入看一下引导加载程序中的C的部分 `boot/main.c` 。

> **练习四：**
>
> 大致回忆一下C的知识，尤其是指针部分。

为了使 `boot/main.c` 有意义，需要知道什么是ELF二进制文件：当编译和链接C程序时，编译器会将每个C源文件转换为包含二进制格式编码的汇编语言指令的.o文件。链接器然后将所有编译的对象文件合并成一个二进制图像，在这种情况下，它是ELF格式的二进制形式，ELF代表Executable and Linkable Format。

6.828中可以把ELF可执行文件看作是一个带有记载信息的头文件，后跟有若干个程序部分，每个程序部分都是连续的一段代码或者数据，用于加载到内存中的特定地址上。引导加载程序不会对代码和数据进行修改，直接将其加载到内存中然后执行。

一个ELF二进制文件以一个固定长度的ELF头为首，后跟有变长的列有每个讲要加载的程序段的程序头。其中ELF头的C定义位于 `inc/elf.h` ，我们关注的程序部分如下：

- `.text` ：程序可执行指令
- `.rodata` ：只读数据，例如C编译器生成的ASCII字符串常量
- `.data` ：程序已初始化数据，例如已初始化的全局变量

链接器计算好程序的内存分布后，就会保留一些空间供未初始化的全局变量使用，存放在内存中紧跟在 `.data` 部分后边的 `.bss` 部分。C要求未初始化的全局变量初始值必须是0，所以ELF二进制文件中无需为 `.bss` 存放内容，而实际上链接器仅仅记录 `.bss` 部分的地址和大小，加载程序或者程序本身必须将 `.bss` 部分置零。

用 `objdump -h obj/kern/kernel` 命令检查内核可执行文件中所有部分的名字、大小和链接地址。除了上述讲的部分，其他在6.828中都不重要。

着重注意 `.text` 部分中的VMA链接地址和LMA加载地址。一个部分的加载地址是其应该加载到内存中的地址；链接地址则是这个部分应该执行的内存地址。链接器将链接地址编码为二进制的方式又很多...（The linker encodes the link address in the binary in various ways, such as when the code needs the address of a global variable, with the result that a binary usually won't work if it is executing from an address that it is not linked for.）

一般地，链接地址和加载地址是一样的，可以用命令 `objdump -h obj/boot/boot.out` 查看引导加载程序的 `.text` 部分。

引导加载程序通过ELF程序头来确定如何加载一个部分。程序头指定了要加载到内存中的ELF对象的哪些部分以及每个部分应该占用的目标地址，可以通过命令 `objdump -x obj/kern/kernel` 来检查程序头。

程序头就列在打印结果的 `程序头` 后边，ELF对象需要被加载的区域标记有 `LOAD` ，其他一些信息也相应给出，例如 `vaddr` 标记的虚拟地址， `paddr` 标记的物理地址， `memsz` 和 `filesz` 标记的加载区域的大小。

回到 `boot/main.c` ，每个程序头的 `ph->p_pa` 字段包含段的目标物理地址（在这种情况下，它实际上是一个物理地址，尽管ELF规范对此字段的实际含义是模糊的）。

BIOS将引导扇区加载到以0x7c00开始的空间内，所以0x7c00就是引导扇区的加载地址，同时也是引导扇区执行的起始地址，故而也是其链接地址。我们通过把 `-Ttext 0x7c00` 传递给 `boot/Makefrag` 中的链接器来设置链接地址，因此链接器将在生成的代码中产生正确的内存地址。

> **练习五：**
>
> 追踪引导加载程序中最开始的几条指令，然后找到第一条如果使引导加载程序链接地址错误就会“崩溃”的指令。然后将其在 `boot/Makefrag` 中的链接地址修改成错误的地址，接着执行 `make clean` 并执行 `make` 重新编译，然后追踪到引导加载程序中观察发生的情况。观察之后再将链接地址修改成正确的地址。

回头看一下内核的加载地址和链接地址。和引导加载程序不同，这两个地址是不相同的，因为内核请求引导加载程序将其加载到内存中的低地址（1MB内）处，但是期望从一个高地址处开始执行。

除了节信息，ELF头文件还有一个重要的区域称作 `e_entry` ，这个区域存有程序入口点的链接地址，即text节中程序应该开始执行的内存地址，可以通过 `objdump -f obj/kern/kernel` 来查看入口点。

总结： `boot/main.c` 中的ELF头文件将内核的每个部分从磁盘读取到内存的加载地址，然后跳转到内核的入口点。

> **练习六：**
>
> 我们可以通过GDB的 `x` 命令来检查内存，其完整语法 `x/Nx ADDRESS` 打印内存ADDRESS处的N个字，需要注意的是，这里的字的大小并非一般标准下的字的大小，在GNU汇编中，一个字为2B。
>
> 重置机器（退出QEMU / GDB然后重新运行），在BIOS进入引导加载程序的那一点处检查0x00100000的8个字的内存，然后再次启动加载程序进入内核。为什么不一样？第二个断点处有什么？

第三部分：内核

这一部分将进一步深入探讨JOS内核。和引导加载程序类似，内核由汇编语言开始以便C可以正常执行。

1.用虚拟内存解决位置依赖问题

引导加载程序的链接地址和加载地址完全相同，但是内核的链接地址和加载地址之间确有着巨大的差异（确保你注意到了这一差异）。

操作系统内核通常在很高的虚拟地址（如0xf0100000）处链接和运行，从而将处理器的低地址空间留存给用户程序使用，这种安排的具体原因将在Lab 2中详细介绍。

大部分机器在0xf0100000地址处没有物理内存空间，所以并不能存放内核。但是可以使用处理器的内存管理硬件将虚拟地址0xf0100000映射到物理地址0x00100000上，这里恰好是引导加载程序将内核加载到内存空间的位置。这样一来，尽管内核的虚拟地址足够高，为用户进程留下了足够的地址空间，但它将被加载到PC RAM内的1MB处的物理内存中，正好在BIOS ROM的上方。这种方法要求PC具有至少几MB的物理内存（从而物理地址0x00100000可以工作），但这只可能在大约1990年之后构建的PC上有效。

实际上，在Lab 2中，我们将会把整个PC物理地址空间底部的256MB（0x00000000到0x0fffffff）映射到虚拟地址0xf0000000到0xffffffff。现在应该可以理解为什么JOS仅可以使用物理内存的前256MB。

现在我们只映射物理内存的前4MB作为开端，通过 `kern/entrypgdir.c` 中手写的、静态初始化的页目录和页表。现在暂时不需要清楚其工作原理，只需要知道可以达到的效果即可。到 `kern/entry.s` 设置 `CR0_PG` 标志以前，内存引用一直被看做物理地址（严格来说，是线性地址，但是 `boot/boot.s` 建立起标识映射，那是永远不会改变的）。一旦 `CR0_PG` 设置完成，内存引用就变成了由虚拟内存硬件转换为物理地址的虚拟地址。 `entry_pgdir` 将0xf0000000至0xf0400000范围内的虚拟地址转换为物理地址0x00000000至0x00400000，以及0x00000000至0x00400000范围内的虚拟地址转换为物理地址0x00000000至0x00400000。任何不在这两个范围内的虚拟地址都将导致硬件异常，从而导致QEMU转储机器状态并退出。

> **练习七：**
>
> 使用QEMU和GDB追踪到JOS内核中，停止到 `movl %eax, %cr0` 。检查0x00100000处和0xf0100000的内存。然后使用GDB命令 `stepi` 单步执行一次，再检查一次0x00100000处和0xf0100000的内存。观察发生了什么。

新的映射建立后，第一条如果映射不到位就无法正常工作的指令是哪一条？注释掉 `kern/entry.S` 中的 `movl %eax, %cr0` ，跟踪到它，看结果是否正确。

2.格式化打印到控制台

在一个OS内核中，所有IO操作都需要我们自己实现。

仔细阅读 `kern/printf.c` ， `lib/printfmt.c` 和 `kern/console.c` ，确保理解了其中的联系。后期的Lab中将会阐述为什么 `printfmt.c` 独自安排在 `lib` 目录下。

> **练习八：**
>
> 找出并补充省略的 `%o` 打印八进制数的代码块

回答下列问题：

1. 解释 `printf.c` 和 `console.c` 之间的接口。具体来说， `console.c` 导出哪个函数？ `printf.c` 又是如何使用这个函数的？
2. 解释 `console.c` 中的代码段：

```
1      if (crt_pos >= CRT_SIZE) {
2              int i;
3              memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
4              for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
5                      crt_buf[i] = 0x0700 | ' ';
6              crt_pos -= CRT_COLS;
7      }
```

3. 下列问题，你可能需要参考Lec 2的笔记，这些笔记涵盖了GCC在x86上的调用规则：

   以步进的方式追踪代码段：
   ```
   int x = 1, y = 3, z = 4;
   cprintf("x %d, y %x, z %d\n", x, y, z);
   ```

   - 调用 `cprintf()` 时， `fmt` 指向的是什么？ `ap` 指向的又是什么？
   - 按执行顺序列出每次对 `cons_putc` ， `va_arg` 和 `vcprintf` 的调用。对于 `cons_putc` ，同时列出其参数；对于 `va_arg` ，列出调用前后的 `ap` 的指向；对于 `vcprintf` 列出其两个参数的值。

4. 运行代码：

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

   输出是什么？说明如何以上一个练习的步进方式完成这个输出。可参考ASCII表。

   上述输出取决于x86是小端模式。如果x86是大端模式，应该如何设置 `i` 以达到相同的输出？是否需要修改值 `57616` ？

5. 在下边的代码段中， `y=` 后将会打印什么（不是一个确定的值）？为什么？ `cprintf("x=%d y=%d", 3);`

6. 假设GCC改变了调用规则从而以声明顺序推入参数。那么如何修改 `cprintf` 或其接口才可以使之仍然可以传递数目可变的参数？

3.栈

Lab 1的最后将会探索以下问题：

1. C在x86上使用栈结构的细节
2. 在此过程中编写一个有用的新内核监视器函数。该函数打印堆栈的回溯：一个来自引导到当前执行点的嵌套 `call` 指令的指令指针（IP）值的列表。

> **练习九：**
>
> 确定内核初始化堆栈的位置以及堆栈所在内存的确切位置。内核如何为堆栈保留空间？以及堆栈指针应该指向该区域的哪一端？

x86栈指针（ESP寄存器）指向的是栈中正在被使用的最低位置，任何低于这个位置的部分都是空闲的。将一个值入栈包含降低栈指针和将值写入栈指针指向的位置两个步骤；将一个值出栈包含从栈指针中读取数值和提高栈指针两个步骤。在32位模式下，栈结构只可以保存32位的值，并且ESP总是可以被4整除。许多x86指令如 `call` ，都是通过硬连线来使用栈指针寄存器的。

x86基指针（EBP寄存器）则恰好相反，主要通过软件规则与栈相关联。一个C函数的入口中，函数的序幕代码通常会把前一个函数的基指针入栈以达到保存的目的。如果一个程序中所有函数都遵循这个规则，那么在这个程序执行的任意给定时刻，都可以通过跟踪保存的EBP指针、准确确定导致将要达到的程序特定点的函数调用的嵌套序列栈，从而追溯到栈。这一功能非常有利，例如，当一个特定的功能因为传递了错误的参数导致了 `assert` 失败或者 `panic` ，但却不能确定是哪一函数传递了错误参数，这时通过栈的回溯即可找到出问题的函数。

> **练习十：**

为熟悉x86下的C调用规则，找到 `obj/kern/kernel.asm` 中的 `test_backtrace` 函数并设置断点，然后观察内核开始后每次调用这个函数会发生什么情况。每个递归嵌套层级的 `test_backtrace` 会将多少个32位字入栈，这些字是什么？

上述练习说明需要实现一个栈回溯函数 `mon_backtrace()` 。 `kern/monitor.c` 中已经提供了这个函数的原型，可以完全用C实现，但是 `inc/x86.h` 中的 `read_ebp()` 可能会有所帮助。不过还需要将这个函数放入内核监视器的命令列表中从而用户也可以调用。

回溯函数应当以如下形式显示一列函数的调用框架：

```
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` 三个项。 `ebp` 值代表被该函数使用的基指针，即紧跟在函数进入和序幕代码设置基指针之后的位置。 `eip` 值表示的是该函数的返回指令指针，即当函数返回时控制将要返回的指令地址。返回指令指针一般指向的是 `call` 指令的下一条指令（为什么？）。最后5个16进制数是存在问题的函数的前5个参数，这些参数在调用之前被送入栈中。如果函数的参数少于5个则某些值是无效的。（为什么回溯代码不能检测到试剂有多少个参数？这个局限是如何被改善的？）

打印的第一行反映了当前正在执行的函数，即 `mon_backtrace` 本身，第二行反映了调用 `mon_backtrace` 函数的函数，第三行反映了调用该函数的函数，依此类推。通过学习 `kern/entry.s` 很容易发现什么时候停止。

练习十一：

实现上述的回溯函数。如果你使用 `read_ebp()` ，注意GCC可能会在 `mon_backtrace()` 函数序幕之前生成调用 `read_ebp()` 的优化代码，这将导致栈追踪的不完整，因为最近的函数调用的栈帧丢失。虽然我们尽量禁止导致此重新排序的优化，但可能需要检查 `mon_backtrace()` 的程序集，确保在函数序幕之后调用 `read_ebp()` 。

至此，回溯函数应该可以给出栈中导致 `mon_backtrace()` 被执行的函数调用的地址。但是实际中可能更需要函数的名字而不是地址，也许想知道是哪一个函数中存在导致内核崩溃的错误。

为实现这一功能，我们已经提供了函数 `debuginfo_eip()` ，该函数从符号表中查询eip，然后返回相应地址的调试信息。该函数定义在 `kern/kdebug.c` 中。

练习十二：

改进回溯函数，使其可以打印每一个eip对应的函数名、源文件名以及所在行。

在 `debuginfo_eip` 中， `__STAB_*` 来源于哪里？这里提供帮助：

- 从文件 `kern/kernel.ld` 中寻找 `__STAB_*`
- 运行 `objdump -h obj/kern/kernel`
- 运行 `objdump -G obj/kern/kernel`
- 运行 `i386-jos-elf-gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno- format -DJOS_KERNEL -gstabs -c -S kern/init.c` ，然后查看 `init.s`
- 观察引导加载程序是否将符号表加载到内存中作为加载内核二进制文件中的一部分

完成 `debuginfo_eip` 的实现，插入对 `stab_binsearch` 的调用以找到地址的行号。在内核监视器中添加 `backtrace` 命令，并将 `mon_backtrace` 的功能扩展至调用 `debuginfo_eip` 为每一栈帧打印一行：

每一行都给出了栈帧eip所在文件名和对应行，后边跟着函数名和函数中eip从第一条指令开始的偏移量，例如 `monitor+106` 表示返回的eip位于monitor开头后的106字节处。

提示： `printf` 提供了一种虽然模糊但是很简单的方法来输出类似STABS表的非空终止的字符串： `printf("%.*s", length, string)` ，可以输出 `string` 中长度不超过 `length` 的部分。查看 `printf` 手册页，找出这为什么有效。

你可能会发现回溯中有一些丢失的函数，比如你可能看到了 `monitor()` 的调用但是没有看到 `runcmd()` 的调用，这是因为编译器将一些函数内联。其他一些类似的优化也可能导致所在行与期望不一致，你可以将 `GNUMakefile` 中的 `-O2` 删去消除问题，但这将导致内核运行变慢。