实验一: 启动PC

介绍

本次的实验分为三个部分，第一部分主要熟悉一下X86汇编语言，QEMU的X86模拟器和PC开机的引导过程。第二部分检查内核的引导加载程序，它位于实验源代码的boot文件夹中。最后，第三部分深入研究内核本身的初始化，相关代码在kernel目录中。在后文中，我们把内核命名为JOS。

软件设置

对于本次实验和接下来的实验课程中所用到的文件，我们使用Git 版本控制器进行版本管理。想了解更多Git相关的信息，请查看Git user's manual , 如果你以前使用过其他的版本控制系统，这里有一个非常有用的GIT的简单介绍CS-Oriented overview of Git。

有关本课程的的Git远程仓库

由于本实验来自MIT的实验课程，但是MIT的服务器不提供对外的服务，因此原文档中关于使用MIT服务器的相关介绍均不可用，相关的内容我们已下载到本地的服务器中。请从以下地址下载资源，解压缩后开始实验：http://oslab.mobisys.cc/docs/lab1.zip

解压缩后，实验相关的代码在lab1/src/lab1_1下

Git允许你跟踪你所修改的代码。例如你完成了一到两个练习，想要检查你的进度，可以通过一下的指令来提交你的一些修改：

```
athena% git commit -am 'my solution for lab1 exercise 9'
Created commit 60d2135: my solution for lab1 exercise 9
 1 files changed, 1 insertions(+), 0 deletions(-)
athena%
```

也可以通过git diff 指令浏览你修改的内容。运行了git diff将会显示从上次提交后你代码中的写改动，git diff orign/lab1 将会展示自从为 初始化代码后写相关的变化。在这里，origin/lab1 git分支，你可以从我们的服务器中下载的这个任务的初始化代码。

你需要根据实验准备的指南安装qemu 和gcc 。我们对qemu 和进行更改，加入了一些有用的调试功能，接下来的一些实验会用到这些修改的补丁，需要实验依赖的补丁因此，因此，你必须在你的机器上编译和安装这一版修改过的qemu，而不是下载qemu的发行版。如果你的机器用来自己的ELF的工具链（例如Linux 和绝大多数的BSD的，但不是OSX）你需要从你的安装包中安装一下gcc编译器，你可以按照工具页面上的说明进行操作。

提交的过程

我们会通过一个评定程序对你的解决方案进行一个评定。你可以通过评定程序并运行make grade 来检测的你的解决方案。

不必运行make handin等其他文档中的命令，那些将会触发向MIT服务器的提交工作，对我们来说是无效的。

Part1：PC引导程序

第一个练习的目的是向你介绍有关X86汇编编程语言和PC的引导进程，并让你了解QEMU和QEMU/GDB 的调试。接下来你并不需要为这个实验写任何的代码，但是为了你能更好的理解这个实验，你必须认真的浏览一些。

X86汇编入门

如果你之前并不怎么熟悉X86汇编语言，通过本课程你将会很快熟悉汇编语言。PC Language Book 是一本非常适合初学者的书。这本书能包含了一些新旧特性，希望这能对你有助。

提示：不巧的是PC Language Book 这本书的例子是用的NASM 汇编器，而我们用的是GNU的汇编器。NASM 用的是所谓的英特尔的语法而GNU使用的是AT&T的语法。即便是表达的是相同的意思，汇编文件也会有很大的不同，至少表面上是这样，不过对于两者之间的转换是非常简单的，两者间的转换在Brennan's Guide to Inline Assembly 深入讲解。

练习 1. 熟悉6.828 reference materials page面上提供的汇编语言的资料。你不需要马上阅读这部分的资料，但是实验的过程中你肯定会用这一部分资料。

我们建议阅读一下 Brenna's Guide to Inline Assembly 的“The Sytax”这一部分的内容。这一部分内容给出了我们将在JOS中与GUN汇编器一起使用的AT&T汇编语法的一个很好的（简洁的）描述。

当然，X86汇编语言的定义是根据英特尔架构的指令集合设计的，你可以在参考资料页面上看到这些内容：一个是80386 Programmer's Reference Manual，这个版本相较于最进的手册目录更加的简短，但详细的描述了我们将要在6.828中使用的所有的X86处理器的特点；最新，最全面英特尔的处理器资料是 IA-32 Intel Architecture Software Developer's Manuals，这里面包含了很多英特尔处理器的新特性，我们课程中不会讲解，但是如果你感兴趣，可以自己去了解。AMD也提供了一套相同的（甚至更好理解的）处理器文档。保留一份AMD或者英特尔架构手册为以后的日常使用，以便在以后的你想了解或者查询某些处理器的特点或者指令方便使用。

模拟X86

相较于实际的在个人PC上开发操作系统，我们用一个模拟PC的程序来进行开发：你为模拟器所编写的代码也可以安装一个真实的PC上。用模拟器可以简化调试；举例来说，你可以在X86模拟器中设置一个断点去调试，单是对于实际X86处理器很难做到。

在实验中我们将使用QEMU模拟器，一个更现代更快的模拟器。虽然QEMU内置了显示器仅提供了有限的调试功能，QEMU 可以当作GNU 调试器的远程调试目标(GDB)，我们将在实验中使用这种方式，来完成早期的启动过程。

开始前，我们解压lab1 中的文件到你的主机里，接下来在lab文件夹下输入make（或者 在BSD的系统下输入gmake）去构建6.828最小的加载启动程序和内核。（现阶段把这些代码叫作“内核”确实有些名不符实，不过接下来的学期中我们将逐步的完善这份代码。）

athena% **cd lab**

athena% **make**
+ as kern/entry.S
+ cc kern/init.c
+ cc kern/console.c
+ cc kern/monitor.c
+ cc kern/printf.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 414 bytes (max 510)
+ mk obj/kern/kernel.img
（如果你在运行过程中得到类似“__udivdi3未定义引用”的错误,你可能是没有32位的gcc multilib．如果你运行在Debian 或者Ubuntu 上，请尝试安装gcc-multilib 的安装包，请参考FAQ页面）

现在，你即将运行QEMU，创建 obj/kern.kernel.img 上面提供的文件作为仿真PC的“虚拟硬盘”的内容。这个硬盘镜像包含了我们需要的启动代码bootloader（obj/boot/boot） 以及我们使用的内核（obj/kernel）

athena% **make qemu**

这条命令将执行QEMU，默认的配置选项是使用编译出来镜像为硬盘，并将串行端口输出显示到终端上。某些文本应显示在QEMU窗口中：

Booting from Hard Disk...
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
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>

从“Booting from Hard Disk...”后所有内容都是从我们的JOS 内核打印出来的； k> 是型监视器提示符，是一个交互式控制程序，已经包含在内核中了。以上这些行也会出现在运行QEMU的常规

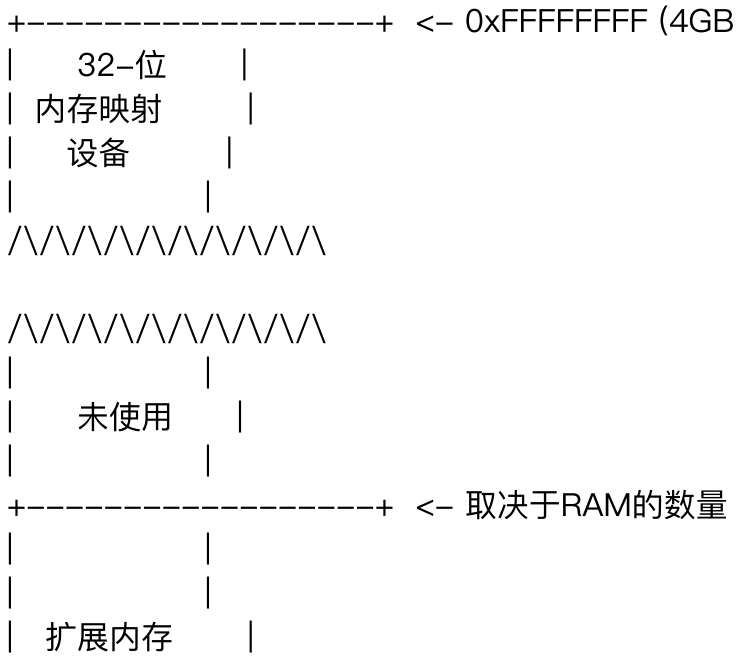
shell窗口中。为了实现自动测试和评分，我们调整了JOS的内核将其控制结果的输出不仅写到了虚拟的VGA显示器中（能在QEMU窗口上可以看见的），也输入到模拟PC的虚拟串行接口上，QEMU将输出转为自己的标准输出，显示在屏幕上。同样的，JOS内核也会接收来自于键盘和其它串行设备的输入，因此你既可以将指令出道VGA的显示器上或在运行运行的这QEMU的中端上。另外，你也可以运行make qemu-nox使用没有VGA显示器的串型控制台。 如果你是用的SSH远程登录的方式连接在服务器上进行实验，这可能很方便。

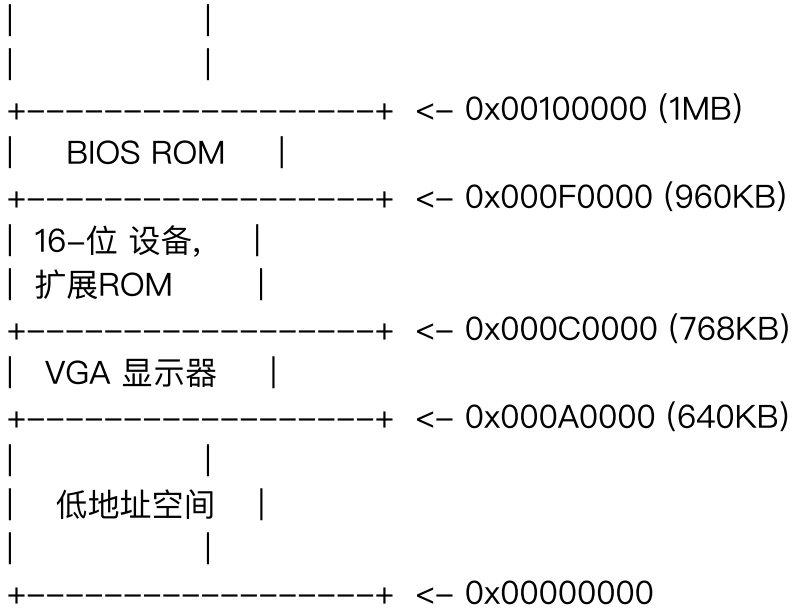
```
现在内核监控器只支持两种命令，help 和 kerninfo
K> help
help - display this list of commands
kerninfo - display information about the kernel
K> kerninfo
Special kernel symbols:
  entry  f010000c (virt)  0010000c (phys)
  etext  f0101a75 (virt)  00101a75 (phys)
  edata  f0112300 (virt)  00112300 (phys)
  end    f0112960 (virt)  00112960 (phys)
Kernel executable memory footprint: 75KB
K>
```

help 命令的作用很明显，我们接下来将简单讨论kerninfo 命令打印内容的含义。即使简单，但是需要注意的是，内核的监控器是“直接”在PC模拟器的“原生（虚拟）的硬件”上运行。 这意味着如果你将obj/kern/kernel.img的内容全部拷贝到硬盘的前几个扇区，将硬盘插入到你的实际的PC机器上，打开它，你会发现在你PC屏幕上的内容和你在QEMU窗口上所做的操作是一样的。（我们不建议你用电脑来操作，一旦你将kernel.img 的内容拷贝到你硬盘前面会修改主引导记录覆盖掉第一扇区的开头部分，这样会导致硬盘上所有东西的都会丢失！）

PC的物理地址空间

现在我们将描述一下PC的启动过程。一个PC的物理地址空间具有以下大致的布局。





第一代PC是基于16位的Intel8088处理器设计的，只能处理1MB的物理内存。因此早期的PC的物理地址空间都是以0x00000000开始，但不是以现在0xFFFFFFFF结束的而是0x000FFFFF。前640KB区域被标记为“低地址空间”，是早期PC唯一可以使用的随机访问存储区（RAM），事实上，早期的PC只配有16KB,32KB,或者64KB的RAM!

从0x000A0000 到0x000FFFFF 的384KB区域被保留下来用作硬件的一些特殊用途，比如视频播放的缓冲区和保存固件的非易失性内存空间。这一部分保留区域中最重要的一部分是基本的输入输出系统（简称BIOS）,占据了从0x000F0000 到 0x000FFFFF这64KB的区域。早期的PC机上BIOS是保存在只读内存区域，现在的PC机上BIOS 存储在新型的闪存存储器上。BIOS负责执行系统的基本初始化，例如激活视频卡或者是检查内存安装的总量。在执行完初始化操作后，BIOS 将从一些合适的地方加载操作系统，例如软盘，硬盘，光盘，或者网络上，然后将机器的控制权交给操作系统。

当英特尔的80286 和80386的处理器最终打破了“一兆字节的瓶颈”，可以支持16MB和4GB的物理地址空间，可是PC的架构师依然要保留原来1MB的低地址的物理空间布局以保证现有软件的向下兼容性。因此，现代PC有一个在0x000A0000到0x00100000 的物理地址上有一部分“空洞”，将RAM划分为两段，“低地址内存空间”（第一个640KB的内存空间）和“扩展内存空间”（其余的部分）。此外，一些在32位机器的物理地址空间（最重要的是物理RAM）最顶端的一些空间，通常被BIOS 保存用作32为的PCI的设备使用。

最近X86的处理器能支持超过4GB的物理内存，因此RAM可以拓展到0xFFFFFFFF之上。在这种情况下BIOS 必须要在32位系统内存高层预留第二个“空洞”，用于为一些32位的设备预留映射空间。由于设计的限制，JOS只能使用前265MB的物理内存，现在我们假设电脑仅有一个连续的32位的物理地址空间。对于复杂的物理空间结构的管理和多种多样的已有设备的兼容，一直是OS不得不面对的一个问题，也是是未来发展中会一直存在的挑战。

BIOS

在这一部分实验中，你将使用QEMU的调试功能来检查一个IA-32兼容的计算机的引导过程。

打开两个终端窗口，再一个里面输入make qemu-gdb(或者是make qemu-nox-gdb)。这个将启动QEMU，但是在QEMU将处理器执行第一个指令前停止，并等待来自GDB的调试连接。在第二

个终端窗口，在运行make那个目录运行gdb，你可以看到像这样的一些信息，

athena% **gdb**

```
GNU gdb (GDB) 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
+ target remote localhost:26000
The target architecture is assumed to be i8086
[f000:fff0] 0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb)
```

我们提供一个.gdbinit 文件，设置GDB调试早期启动时的16位代码和并指示他去监听QEMU。（如果不起作用，你可以尝试在home目录里.gdbinit 文件中插入set auto-load safe-path / 让gdb能够处理我们在.gdbinit 文件中加入的东西。gdb 会询问你是否要执行这条指令）

注意这一行:

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

这是GBD对要执行的第一条指令拆解。从这个输出我们可以总结出以下几件事情:

(1)IBM 的PC 是从物理地址是0x000ffff0的内存位置开始执行的，这是为ROM BIOS 保留的64KB的区域的最顶端。

PC开始执行时cs = 0xf000 和 IP = 0xfff0

(2)第一条被执行的指令是jmp 指令，跳转到分区地址是cs = 0xf000 和IP = 0x305b

为什么QEMU 是这样开始的？这是完全按照英特尔设计的8088处理器工作的，也就是模拟了IBM的PC机上的处理器工作模式。因为在PC中， BIOS 是通过“硬件接线”连接到物理地址是0x000f0000-0x000fffff这段范围上，这种设计能确保BIOS在电脑后充电后开机或者是任何操作系统重启后立刻获取到机器的控制权。这十分的重要，因为重启后RAM中不会有任何的软件可供处理器执行。QEMU 模拟器有自己的BIOS，它位于处理器模拟的物理地址空间的位置上。在处理器复位的时候，（模拟）处理器进入实模式将CS设置到0xf000并且将IP设置到0xfff0 ,以便从（CS：IP）地址开始执行。分段机制地址0xf000:fff0如何转为物理地址的呢?

为了回答这个问题，我们必须了解一些实际模式寻址的知识。在实际模式（PC的启动模式），地址翻译是根据公式： 物理地址 = 16 * 段 + 偏移。因此。当将PC 的CS设置为0xf000 IP设置为0xfff0,物理地址就是:

```
  16 * 0xf000 + 0xfff0   #  在16进制下乘以16
= 0xf0000 + 0xfff0       # 只追加一个0
= 0xffff0
```

0xffff0 是一个BIOS（0x100000）最后16个字节的地址。因此， BIOS做的第一件事是将jmp倒退到在BIOS较前的位置。毕竟在16字节中能完成的工作是很有限的

练习 2. 使用GDB的si (Step Instruction)指令来跟踪ROM BIOS 中的几个指令，并且尝试这些指令是要做什么的。你可能需要查看Phil Storrs I/O Ports Description,以及6.828 reference

materials page 其他的资料。并不需要弄清楚所有的细节–仅需要先弄清楚BIOS工作的一个思路。

当BIOS 运行起来，他建立了一个中断描述符表，并初始化例如VGA设备各种变量。这就是为什么会从QEMU窗口上看到“Starting SeaBIOS” 信息。

当初始化了PCI 总线和所有BIOS知道的重要设备后。他开始搜索启动设备例如软盘，硬盘，光盘等。最终，当找到了一个启动磁盘，BIOS就开始从磁盘里面读取启动加载程序，并将电脑的控制权转让给启动程序。

Part2：引导加载程序

PC的软盘和硬盘以512个字节为单位划分成区域，每个区域称为一个扇区。扇区是磁盘的最小访问单位：每个读、写操作的大小必须是一个或多个扇区，并在扇区边界上对齐。如果磁盘是可引导的，则第一个扇区称为引导扇区，因为这是引导加载程序代码所在的位置。当BIOS找到可启动的软盘或硬盘时，将512字节的引导扇区加载到物理地址0x7c00到0x7dff的内存中，然后使用jmp指令将CS：IP设置为0000：7c00，将控制权传递给引导程序。引导程序又称为bootloader。像BIOS加载地址一样，这些地址是随意设定的 – 但对于PC它们是固定和标准化的。

在PC的演进过程中，从CD-ROM启动的能力要晚很多，因此，PC架构师借此机会重新思考引导过程。因此，现代BIOS从CD-ROM启动的方式有点复杂（且更强大）。CD-ROM使用2048字节大小的扇区，而不是512字节，并且在将控制转移给它之前，BIOS可以将更大的启动映像从磁盘加载到内存（而不仅仅是一个扇区）中。有关详细信息，请参阅 "El Torito" Bootable CD-ROM Format Specification。但是，对于6.828，我们将使用传统的硬盘启动机制，这意味着我们的引导加载程序必须放进512字节的区域中。引导加载程序由两个文件组成，一个汇编语言源文件boot/boot.S和一个C语言源文件boot/main.c。仔细阅读这些源文件，并确保你了解发生了什么。引导加载程序必须执行两个主要功能：

首先，引导加载程序将处理器从实模式切换到32位保护模式，因为只有在该模式下，软件才能访问处理器物理地址空间中1MB以上的所有内存。保护模式在 PC Assembly Language第1.2.7和1.2.8节中有简要介绍，并在英特尔架构手册中有详细介绍。在这个阶段上，你只需要理解，在保护模式下，将分段地址（段地址：偏移量的组合）转换为物理地址的方式不同，而转换后的偏移量为32位而不是16位。

其次，引导加载程序通过x86的特殊I/O指令直接访问IDE磁盘设备寄存器，从硬盘读取内核。如果你想更好地了解这里的特定I/O指令是什么意思，请查看the 6.828 reference page “IDE硬盘驱动器控制器(IDE hard drive controller)”部分。在本课程中，你不需要了解编程特定设备的情况：编写设备驱动程序实际上是操作系统开发中非常重要的一部分，但从概念或架构的角度来看，它也是最无趣的一部分。

在了解了引导加载程序的源代码后，查看文件obj/boot/boot.asm。该文件是我们的GNUmakefile在编译引导加载程序之后创建的引导加载程序的反汇编代码。从这个反汇编文件中可以很容易地查看所有引导加载程序代码所在的物理内存的确切位置，并且可以更轻松地监视GDB中的引导加载程序中发生的情况。同样，obj/kern/kernel.asm中包含的是JOS内核的反汇编结果，这通常对调试很有用。

你可以使用b 命令在GDB中设置地址断点。例如，b *0x7c00设置地址0x7C00的断点。一旦执行停在了断点处，可以使用继续执行c与 si命令：c使QEMU继续执行，直到下一个断点（或者直至你在GDB按下 Ctrl-C）；si N的意思是连续执行N条指令后停下来。

要检查内存中的指令（GDB会自动打印出马上要执行的下一个指令，但并不会显示其他指令），请使用该 x/i命令。该命令的语法是 x/Ni ADDR，其中N是要反汇编的连续指令的数量，ADDR是要开始反汇编的存储器地址。

练习3. 查看lab tools guide，特别是GDB命令部分。即使您熟悉GDB，这包括一些对操作系统工作非常有用且深奥GDB命令。

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

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

下面回答以下问题：

处理器什么时候开始执行32位代码？如何完成的从16位到32位模式的切换？

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

内核的第一个指令在哪里？

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

加载内核

现在我们将进一步详细介绍在boot/main.c中bootloader程序的C语言部分。在这之前，需要回顾一些C编程的基础知识。

练习4. 阅读C中的指针编程.C语言的最佳参考是 Brian Kernighan和Dennis Ritchie（被称为“K＆R”）的C编程语言（中文翻译版）。

通读K＆R书中5.1（指针和地址）到5.5（字符指针和函数）的内容。然后下载pointers.c的代码，运行它，并确保你了解所有打印值的来源。特别是，请确保你理解第1行和第6行中指针指向哪里的地址，第2行到第4行的值是如何被写入的，以及为什么第5行中打印的值看起来像是错乱的。

还有其他有关C指针的参考书（例如，A tutorial by Ted Jensen，其中大量引用了K＆R的内容）。

警告： 除非你已经彻底地掌握C，否则不要跳过这段阅读练习。如果你真的不了解C中的指针，那么在后续的实验中你将会遭受无法忍受的痛苦，最后会很难理解他们。

为了能够理解boot/main.c，你需要知道什么是ELF二进制文件。当你编译和链接C程序（如JOS内核）时，编译器会将每个C源文件（“.c”）转换为对象文件（“.o”），其中包含了以硬件预期的二进制格式编码的汇编语言指令。链接器随后将所有编译的对象文件合并成一个二进制镜像文件，如obj/kern/kernel，在这种情况下，我们把他叫作是ELF格式的二进制形式，代表Executable and Linkable Format （可执行和可链接格式）。

有关此格式的完整信息可在 reference page上的the ELF specification获得，但你不需要深入了解此类中此格式的详细信息。虽然整体格式是相当强大和复杂的，但大部分复杂的部分是为了支持动态加载共享库，我们不会在作业中涉及这部分内容。在 Wikipedia page有一个简短的描述。

如果只是为了完成本实验，你可以简单的将ELF可执行文件理解为具有加载信息的文件头部，后跟几个程序片段，每个程序片段都是连续的代码或数据块，用于加载到内存中指定的地址处。引导加

载程序不会修改代码或数据; 它将其加载到内存中并开始执行它。

ELF二进制文件以固定长度的ELF头开始，后面是一个可变长度的程序头，程序头中列出了要加载的每个程序段。这些ELF头的C定义在inc/elf.h中。我们感兴趣的程序部分是：

- .text：程序的可执行指令。
- .rodata：只读数据，如C编译器生成的ASCII字符串常量。（但是，我们专门设置硬件来禁止写操作。）
- .data：数据部分保存程序的初始化数据，例如用int x = 5初始化时声明的全局变量; 。

当链接器计算程序的存储器布局时，它会为未初始化的全局变量保留空间（例如int x;这种变量），这些变量会放在一个名为.bss的段中，紧跟着.data的后面。C要求“未初始化”的全局变量自动被初始化为0，因此，不需要存储.bss的，链接器只记录了.bss段的地址和大小。加载程序或程序本身必须将.bss部分置零。

通过键入以下内容，检查内核可执行文件中所有部分的名称，大小和链接地址的完整列表：

athena% **i386-jos-elf-objdump -h obj/kern/kernel**

如果你的计算机像大多数现代Linuxen和BSD一样默认使用ELF工具链，你可以用objdump替换i386-jos-elf-objdump。

你会看到比上面列出的更多部分，但其他部分对我们的目的来说并不重要。其他的大多数都是保存调试信息，这通常包含在程序的可执行文件中，但不由加载程序加载到内存中。

特别注意.text部分的“VMA”（或链接地址link address）和“LMA”（或加载地址load address）。每部分的加载地址是将该部分加载到内存中的内存地址。

段的链接地址是该段期望执行的内存地址。链接器在生成的二进制程序中大量使用链接生成的绝对地址，例如当代码中需要某个全局变量的地址时会直接使用由链接器生成的变量的链接地址，但这样做的结果是如果二进制文件没有被加载到链接地址，而是从一个地址开始执行时，它会产生很多链接地址错误。（生成不包含任何这样的绝对地址的位置无关代码也是可以做到的，这种技术在现代共享库中广泛使用，但它当然也会引入性能和复杂性的代价，所以我们不会在本实验中使用它）。

通常，链接和加载地址是相同的。例如，查看引导加载程序的.text部分：

athena% **i386-jos-elf-objdump -h obj/boot/boot.out**

bootloader程序使用ELF 程序头来决定如何加载这部分。程序头指定要加载到内存中的ELF对象的段的位置以及每个段应该占用的目标地址。你可以通过键入以下内容检查内核的程序头：

athena% **i386-jos-elf-objdump -x obj/kern/kernel**

然后程序头在objdump的输出中的“Program Headers”下列出。需要加载到内存中的ELF对象的区域是标记为“LOAD”的区域。它还提供了每个程序头的其他信息，例如虚拟地址（“vaddr”），物理地址（“paddr”）和加载区域的大小（“memsz”和“filesz”）。

下面我们回到boot/main.c的代码分析中，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规范对该字段的实际含义是模糊的）。

BIOS将引导扇区加载到内存地址0x7c00处，这个地址是引导扇区的加载地址。这也是引导扇区开始执行的地方，所以这也是它的链接地址。我们通过在boot/Makefrag中利用参数-Ttext 0x7C00传递给链接器，以设置链接地址。有了这个配置，链接器将在生成的代码中产生正确的内存地址。

练习5. 跟踪bootloader程序的前几个指令，找到开始使用链接地址的第一条指令，即，如果你使用了错误的链接地址，那么执行到这里的时候就必须要停下来，否则就会发生错误。然后将boot/Makefrag中的链接地址更改为一个错误的地址，运行make clean，用make命令重新编译实

验，然后再次跟踪到引导加载程序，看看会发生什么。不要忘了改变链接地址后要再次执行make clean！

再来看一下内核的加载和链接地址。与bootloader程序不同，这两个地址不一样：内核告诉引导加载程序将其加载到低地址（1兆字节）的内存中，但它希望从高地址执行。我们将在下一节中介绍我们如何实现这一过程的。

除了段信息之外，ELF标题中还有一个对我们很重要的字段，叫做e_entry。该字段保存程序中入口点的链接地址：在程序的代码段中，开始执行的第一条语句在内存中的位置地址。你可以通过以下命令看到入口点：

athena% i386-jos-elf-objdump -f obj/kern/kernel

你现在应该能够了解boot/main.c的意义，这是一个最简单的ELF加载程序。它将内核的每个部分从磁盘读取到存储区的加载地址，然后跳转到内核的入口点。

练习6. 我们可以使用GDB的x命令检查内存。 GDB manual 中详细讲解了命令的用法，但就目前而言，知道命令x/Nx ADDR能够在ADDR处打印N个存储器字就足够了。（请注意，命令中的“ x ”均为小写。） 警告：字的大小不是通用标准。在GNU程序集中，一个字是两个字节（xorw中的'w'，代表字，表示2个字节）。

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

Part3:内核

现在我们将开始更详细的了解JOS内核(你终于可以写一些代码了)。像bootloader程序一样，内核开始于一些汇编语言代码，用于设置事物，以便C语言代码可以正常执行。

使用虚拟内存解决位置依赖问题

当你检查上面的bootloader程序的链接和加载地址时，它们是完全匹配的，但内核的链接地址（由objdump打印）与其加载地址之间存在一个（相当大的）差异。回去检查这两个概念，确保你可以看懂我们在说什么。 （链接内核比引导加载程序更复杂，链接和加载地址位于kern/kernel.ld的顶部。）

操作系统内核通常喜欢链接并运行在非常高的虚拟地址（例如0xf0100000），以便留下处理器虚拟地址空间的较低部分供用户程序使用。 在下一个实验中，这种安排的原因将变得更加清晰。

许多机器在地址0xf0100000没有任何物理内存，所以我们不能指望能够在那里存储内核。 相反，我们将使用处理器的内存管理硬件将虚拟地址0xf0100000（内核代码期望运行的链接地址）映射到物理地址0x00100000（引导加载程序将内核加载到物理内存中的位置）。 这样，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的物理内存中，就在BIOS ROM的上方。 这种方法要求PC具有至少几兆字节的物理内存（从而物理地址0x00100000的位置是是有可用的物理内存的），在1990年左右之后，任何一台PC都是这样的。

实际上，在下一个实验中，我们将把256MB PC的物理地址空间从物理地址0x00000000到0x0fffffff映射到虚拟地址0xf0000000到0xffffffff。 你现在应该会看到为什么JOS只能使用第一个256MB的物理内存。

现在，我们将映射前4MB的物理内存，这足以让我们开始运行。我们在kern/entrypgdir.c中使用手写的，静态初始化的页面目录和页面表来执行此操作。在本实验中，你不必了解这个工作的细节，只需要知道它实现的效果。直到kern/entry.S设置CR0_PG标志，内存引用被视为物理地址（严格来说，它们是线性地址，但是boot/boot.S中把线性地址和物理地址设置成了一致的，我们后面也不会再改变这个配置了）。一旦设置了CR0_PG，内存引用就变成了虚拟地址，虚拟内存硬件转换为物理地址。entry_pgdir将0xf0000000至0xf0400000范围内的虚拟地址转换为物理地址0x00000000至0x00400000，同时也将虚拟地址0x00000000至0x00400000转换为物理地址0x00000000至0x00400000。任何不在这两个范围之一的虚拟地址将导致硬件异常，因为我们还没有设置中断处理，将导致QEMU打印出机器状态并退出（或者如果你没有使用实验中带补丁的QEMU则会无休止地重新启动）。

练习7.使用QEMU和GDB跟踪到JOS内核并停止在movl%eax，%cr0。 查看内存中在地址0x00100000和0xf0100000处的内容。 下面，使用GDB命令stepi单步执行该指令。 指令执行后，再次检查0x00100000和0xf0100000的内存。 确保你明白刚刚发生的事情。

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

格式化打印到控制台

大多数人认为像printf（）这样的功能是理所当然的，有时甚至将它们视为C语言的“原始本能”。 但是在操作系统内核中，我们必须自己实现所有的I/O。

阅读kern/printf.c，lib/printfmt.c和kern/console.c，并确保你了解他们的关系。以后的实验室会清楚，为什么printfmt.c位于lib目录中。

练习8.我们省略了一小段代码 – 使用“%o”形式的模式打印八进制数字所需的代码。查找并补全此代码片段。

你可能会需要参考以下内容：

1) 用 va_list 可以定义一个 va_list 型的变量，这个变量是指向参数的指针。

2) 用 va_start 宏可以初始化一个 va_list 变量，这个宏有两个参数，第一个是 va_list 变量本身，第二个是可变的参数的前一个参数，是一个固定的参数。

3) 用 va_arg 宏可以返回可变的参数，这个宏也有两个参数，第一个是 va_list 变量， 即指向参数的指针，第二个是我们要返回的参数的类型。

4) 最后我们还可以用 va_end 宏结束可变参数的获取

lpt-putchar 和 outb 函数跟设备操作有关，不必细化了解。

格式变量：

padc、width、precision、lflag、altflag。padc 代表的是填充字符，在初始化的时候 padc 变量会被初始化为空格符，而当程序在显示字符串的’%’字符后读到’ –’或者’0’的字符时便会将’ –’或者’0’赋值给 padc。width 代表的是打印的一个字符串或者一个数字在屏幕上所占的宽度，而

precision 则特指一个字符串在屏幕上应显示的长度，当 precision 大于字符串本身长度时相当于 precision 就等于字符串长度。于是在显示字符串的时候 precision 小于 width 的部分则由之前所说的填充字符 padc 来补充，如果 width 小于 precision 则字符串的宽度就等于 precision，而 precision 得默认值-1 代表显示长度为字符串本来的长度。当打印字符串的时候，padc='-'代表着字符串需要左对齐，右边补空格， padc=' '代表字符串右对齐，而左边由空格补齐，padc='0'代表字符串右对齐，左边由 0 补齐。在我们这个实验中当输出数字时会一律的右对齐，左边补 padc，数字显示长度为数字本身的长度。lfag 变量则是专门在输出数字的时候起作用，在我们这个实验中为了简单起见实际上是不支持输出浮点数的，于是 vprintfmt 函数只能够支持输出整形数，，输出整形数时，当 lflag=0 时，表示将参数当做 int 型的来输出，当 lflag=1 时，表示当做 long 型的来输出，而当 lflag=2 时

表示当做 long long 型的来输出。最后，altflag 变量表示当 altflag=1 时函数若输出乱码则用'?'代替。

能够回答以下问题：

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

2.从console.c解释以下内容：

```
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.对于以下问题，你可能需要参考一些课外资料。

跟踪以下代码的并单步执行：

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

- 在调用cprintf（）时，fmt是什么意思？ ap是什么意思?
- 列出（按执行顺序）以下每次调用cons_putc，va_arg和vcprintf这三段代码时的细节。对于cons_putc，列出其参数。 对于va_arg，列出调用之前和之后的ap指针的指向。 对于vcprintf列出其两个参数的值。

4.运行下面的代码

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

输出是什么？如何按照上一个练习的执行步骤，说明为什么会显示这个输出信息。可以参考将字节映射到字符的ASCII表。

由于x86是小端的，所以得到了上面的输出结果。如果x86是大端，那么为了产生相同的输出，你会设置什么？你需要将57616更改为不同的值吗？这是一个关于大小端的描述，和这里是一个更加脑洞大开的说法。

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

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

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

挑战 增强控制台的能力以允许以打印不同颜色的文本。传统的方法是使它解析嵌入在待打印的文本字符串中的ANSI转义序列，但是你可以使用任何你喜欢的机制。 在实验的参考内容和网络上其他地方有对VGA显示硬件进行编程的大量信息。 如果你真的喜欢挑战，也可以尝试将VGA硬件切换到图形模式，并使控制台将文本绘制到图形帧缓冲区上。

栈

在本实验的最后一个练习中，我们将更详细地探讨C语言在x86上使用栈的方式，并编写一个新的内核监视器函数，该函数能够实现栈的回溯打印：一个保存引导到当前执行点的嵌套调用指令的指令指针（IP）值的列表。

练习9.确定内核在哪里完成了栈的初始化，以及栈所在内存的确切位置。内核如何为栈保留空间？栈指针初始化时指向的是保留区域的“哪一端”。

x86栈指针（[Gong1] esp寄存器）指向栈当前正在使用的最低地址位置。所有低于这个位置的栈保留区域都是空闲的。将值压入到栈时，首先会将栈指针的值减少，然后将值写入栈指针指向的位置。 从栈中弹出一个值，需要读取栈指针指向的值，然后增加栈指针。 在32位模式下，栈只能保存32位值，而esp总是可以被4整除。各种x86指令（如函数调用）都是与硬件绑定的，会固定使用栈指针寄存器来操作数据。

ebp（基地址指针）寄存器与栈的关联关系来自于软件设计的传统。在进入C函数时，函数的前导代码通常会将之前的函数的基地址寄存器（ebp）压入堆栈，然后在函数的运行过程中会将当前的esp值复制到ebp中。 如果程序中的所有函数都遵循这一约定，那么在程序执行期间的任何给定点，可以通过遵循保存的ebp指针链来精确地还原出整个ebp的保存链条，并用于确定出整个函数的调用序列。这种特性特别有用，例如，因为传递错误的参数，一个特定的函数导致assert失败或panic，但你不确定谁传递了错误的参数。 栈回溯可以让你找到有问题的函数。

练习10.要熟悉x86上C语言函数的调用约定，请在obj/kern/kernel.asm中找到test_backtrace函数的地址，在其中设置一个断点，并检查在内核启动后每次这个函数被调用时会发生什么。 每一级的test_backtrace在递归调用时，会在栈上压入多少个32位的字，这些字的内容是什么?

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

上述练习应该能够提供实现栈回溯函数所需的信息，我们将它命名为mon_backtrace（）。 这个函数的原型已经在kern/monitor.c中定义好了， 你可以完全在C语言中实现它，但你可能会用到inc/x86.h中的read_ebp（）函数。 你还必须将这个新功能挂接到内核监视器的命令列表中，以便用户可以通过命令交互式地调用它。

回溯功能应显示以下格式的功能调用列表：

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

打印的第一行反映了当前正在执行的函数，即mon_backtrace本身，第二行反映的是调用mon_backtrace的函数，第三行反映了调用该函数的函数，依此类推。 你应该打印所有的堆栈帧。通过学习kern/entry.S你会发现有一个简单的方法来告诉你这个链条的回溯应该到什么时候停止。

每行包含一个ebp，eip和args。ebp值表示的是该函数在栈中的基地址，即刚刚进入到函数的代码时栈指针的位置，在函数的前导代码中完成这个指针的设置。列出的eip值是函数的返回指令指针，即当函数返回时控制流会返回到的指令地址。返回指令指针通常指向call指令后的指令（为什么？）。 最后，args之后列出的5个十六进制值是当前函数的前5个参数，这些参数会在函数调用之前被压入到栈中。 如果函数被调用的参数少于5个，那么当然并不是所有这5个值都是有意义的。（为什么回溯代码不能检测到实际有多少个参数？用什么方法可以改进这个缺陷？）

以下是你在K&R的书第5章的阅读时应注意到的几个具体要点，为了接下来的联系和以后的实验，你应该记下来。

- 如果int * p =（int *）100，则（int）p + 1和（int）（p + 1）是不同的数字：第一个是101，但第二个是104.当向指针添加一个整数时， 在第二种情况下，整数默认会乘以指针指向的对象的大小。
- p [i]被定义为与*（p + i）相同，指的是p指向的存储器中的第i个对象。上述的规则在对象大于一个字节时是非常需要注意的。
- &p [i]与（p + i）相同，是p指向的存储器中的第i个对象的地址。

虽然大多数C程序不需要在指针和整数之间进行转换，但操作系统经常这么做。 每当你看到一个内存地址的加法操作时，需要想清楚是一个整数加法还是一个指针加法，并确保要相加的值是不是要乘以每个对象的大小。

练习11.实现如上所述的回溯功能。 请使用与示例中相同的格式，否则打分脚本将会出错。当你认为你的工作正确的时候，运行make grade来看看它的输出是否符合我们的打分脚本所期待的，如果没有，修正发现的错误。在你成功提交实验1的作业后，欢迎你以任何你喜欢的方式更改回溯功能的输出格式。

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

此时，你的backtrace函数应该为你提供调用mon_backtrace（）的函数在栈上的地址。但是，在实践中，你经常想知道与这些地址对应的函数名称。 例如，你可能想知道哪些函数可能包含导致内核崩溃的错误。

为了帮助你实现此功能，我们提供了函数debuginfo_eip（），该函数能够在符号表中查找eip并返回该地址的调试信息。这个函数在kern/kdebug.c中定义。

练习12.修改你的堆栈回溯功能，为每个eip显示与该eip对应的函数名称，源文件名和行号。

在debuginfo_eip中__STAB_ *来自哪里？这个问题的答案可能很长。为了帮助你找到答案，以下是你可能想要做的一些事情：

- 在文件 kern/kernel.ld中查找 __STAB_*
- 运行i386-jos-elf-objdump -h obj/kern/kernel
- 运行i386-jos-elf-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，并查看init.s

- 看看bootloader是否在内存中加载了符号表，作为加载内核二进制文件的一部分

通过插入调用stab_binsearch来查找地址的行号，补全debuginfo_eip的实现。

向内核监视器添加一个backtrace命令，并扩展你的mon_backtrace的实现，调用debuginfo_eip并为每个堆栈框架打印一行：

```
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>
```

每一行给出栈帧的eip，以及这个代码所在的文件的文件名和行，后面是函数的名称和eip与函数的第一条指令的偏移（例如，monitor+ 106表示返回eip是从monitor函数开始的第106字节）。

确保在单独的行上打印文件和函数名称，以避免混淆打分脚本。

提示：printf格式的字符串提供了一种简单的方法来打印非空终止的字符串，如STABS表中的字符串。 printf（“%.* s”，length，string）打印长度最长为length的字符串。 看看printf手册页，找出这一段的工作原理和使用方法。

你可能会发现回溯中少了一些函数。 例如，你可能会看到对monitor（）的调用，但不会调用runcmd（）。 这是因为编译器内联一些函数调用。 其他优化可能会导致你看到的行号和代码中的不太一样。 如果你从GNUMakefile中删除-O2，那么回溯可能会更容易理解一些（但内核运行速度会更慢）。

这就完成了本次实验。在实验目录中，使用git commit提交更改。

[Gong1]