**哈尔滨工业大学计算机科学与技术学院**

**2017年秋季学期《操作系统》**

**实验报告**

|  |  |  |
| --- | --- | --- |
| **姓名** | **学号** | **联系方式** |
| **李俊儒** | **1153710105** | **18800427105** |

# 一、xv6实验系统简述

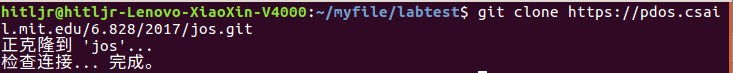
（正文采用宋体四号字，行距1.25倍，段前0行，段后0行。）

xv6 是 MIT 开发的一个教学用的完整的类 Unix 操作系统。它通过研究一个名为 xv6 的操作系统内核来解释操作系统中的主要概念。xv6 是 Dennis Ritchie 和 Ken Thompson 合著的 Unix Version 6（v6）操作系统的重新实现。xv6 在一定程度上遵守 v6 的结构和风 格，但它是用 ANSI C 实现的，并且是基于 x86 多核处理器的。

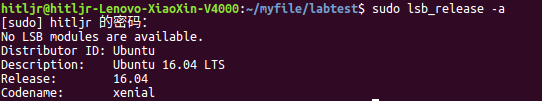
# 二、xv6环境配置（附过程截图）

1. 拷取源代码

这是mit提供的最新的实验平台的实验代码。

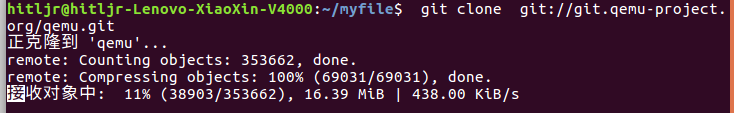
 <https://pdos.csail.mit.edu/6.828/2017/jos.git>

1. 安装linux系统

 我电脑安装的双系统，我的系统版本信息为。

1. 安装GDB

很简单，不再截图。

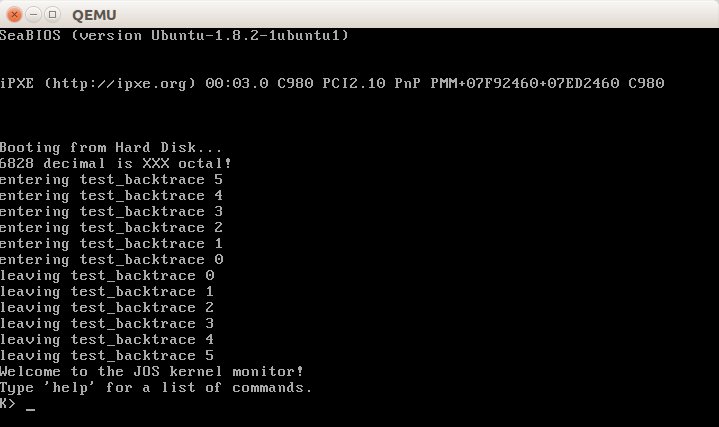
1. 安装QEMU仿真环境

 进入之后，运行./configure

之后make 然后 make install 就大功告成

1. 5.测试以上安装过程

1).在lab文件夹下输入 make

 2).试试qemu，输入 make qemu

如图

图上在 Booting from Hard Disk… 之后的都是我们的JOS内核输出的，K>是小型监视器或者说交互式控制程序的提示。

# 三、实验过程（lab1~lab6）

## Lab 1: Booting a PC

### 实验目的

该实验份为三个部分。

第一部分：

X86汇编语言，QEMU x86仿真器，PC开机引导过程

第二部分：

检查6.828内核的引导加载程序（位于实验树的引导部分）

第三部分：

研究一下JOS

### 实验内容（附过程截图）

熟悉x86、开机引导过程、熟悉使用QEMU、QEMU/GDB调试。不需要写任何代码，但是需要按照整个流程体会并且回答问题。

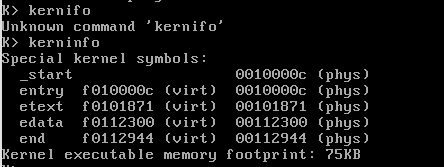
1. 练习1：学习x86:
2. 练习2：模拟x86

模拟工具 QEMU，很先进的模拟工具，并且可以作为GDB取调试。即环境配置的第五步。

JOS内核将从键盘和串行端口输入，所以在VGA显示窗口或者QEMU终端上都可以输入命令。

PS：可以用make qemu-nox来取消VGA窗口显示。

退出QEMU方法：Ctrl+a x

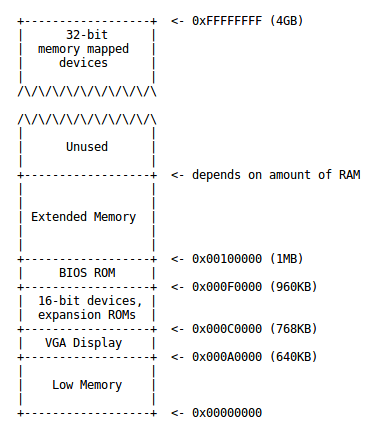
你可以输入两个指令help和kerninfo：

help 展示所有指令和描述

kerninfo 展示内核信息

关于

操作系统的物理地址空间

 总体布局：

简要介绍：

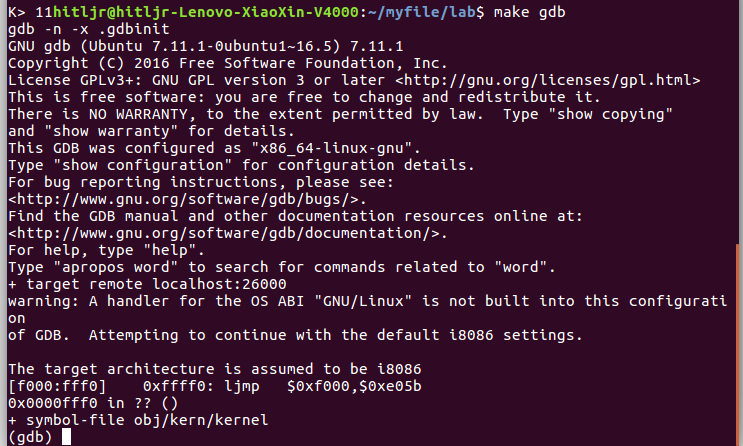
低地址为是RAM。

从0x000A0000到0x000FFFFF的384KB区域被硬件保留，用于诸如视频显示缓冲区和保存在非易失性存储器中的固件等特殊用途。

输入输出系统，早期存在于ROM中，现在位于闪存中，负责系统的初始化

由于很多限制，我们假设JOS只能使用PC的第一个256MB的物理内存，现在假设所有PC只有一个32位的物理地址空间。

输入输出系统

我们在lab文件夹下打开两个终端，在一个中输入make qemu-gdb 然后我们会看到qemu停住了，我们不能进行操作，然后在另一个终端下输入 make gdb 如下图

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

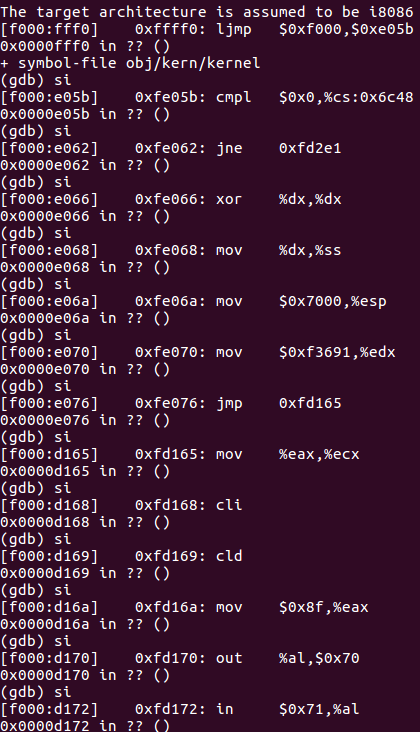
表示第一条指令的反汇编代码，我们可以看出一些东西：

IBM的计算机开始在物理地址0x000ffff0处执行，这个位于输入输出系统保留ROM的顶部

第一条为跳转指令，跳转导CS = 0xf000 和 ip = 0xe05b的位置

这么奇怪的原因是：

因为BIOS地址高，总是在供电或重启之后，首先控制机器.

我们使用si查看下面汇编语句的内容

分别作出如下解释：

1).0xffff0: ljmp $0xf000, $0xe05b

这是运行的第一条指令，是一条跳转指令，跳转到0xfe05b地址处。至于要知道这个地址是怎么通过指令中的值计算出来的.

我们需要先知道，当PC机启动时，CPU运行在实模式(real mode)下，而当进入操作系统内核后，将会运行在保护模式下(protected mode)。实模式是早期CPU，比如8088处理器的工作模式，这类处理器由于只有20根地址线，所以它们只能访问1MB的内存空间。但是CPU也在不断的发展，之后的80286/80386已经具备32位地址总线，能够访问4GB内存空间，为了能够很好的管理这么大的内存空间，保护模式被研发出来。所以现代处理器都是工作在保护模式下的。但是为了实现向后兼容性，即原来运行在8088处理器上的软件仍旧能在现代处理器上运行，所以现代的CPU都是在启动时运行于实模式，启动完成后运行于保护模式。

BIOS就是PC刚启动时运行的软件，所以它必然工作在实模式。

这里先简单介绍一下地址的计算方法，在实模式下，指令中出现的地址都是采用(段基址：段内偏移)的形式的。其中这两个字段的值，通常是存放寄存器中的。其中段基址必须放在段寄存器中，包括CS(代码段), DS(数据段), SS(堆栈段), ES(扩展段)。不同的段寄存器存放的是你程序不同的段的起始地址。但是由于8088CPU中寄存器都是16位，而CPU地址总线是20位的，我们怎么通过16位的寄存器去拼接20位的地址呢？

　　所以我们需要采用下面的方法：把段寄存器中的值左移4位，形成20位段基址，然后和16位段内偏移相加，就得到了真实地址。比如上面的指令中段寄存器的内容为0xf000，所以真实地址为 0xf000<<4+0xe05b = 0xfe05b。

2). 0xfe05b: cmpl $0x0, $cs:0x6ac8

　　下一条指令，把0x0这个立即数和$cs:0x6ac8所代表的内存地址处的值比较，至于为什么这样比较，现在还不是很清楚。其中$cs:0x6ac8就是我们刚刚介绍的在实模式下地址形成的格式，其中$cs就代表CS段寄存器的值。

3）. 0xfe062: jne 0xfd2e1

　　 jne指令：如果ZF标志位为0的时候跳转，即上一条指令cmpl的结果不是0时跳转，也就是$cs:0x6ac8地址处的值不是0x0时跳转。

4-9).

0xfe066: xor %dx, %dx

0xfe068: mov %dx %ss

0xfe06a: mov $0x7000, %esp

0xfe070: mov $0xf34d2, %edx

0xfe076: jmp 0xfd15c

0xfd15c: mov %eax, %ecx

设置一些寄存器的值

10）.0xfd15f cli

关闭中断，启动的时候不能被中断。

11）.0xfd160

设置方向标识位为0，表示后续的串操作比如MOVS操作，内存地址的变化方向，如果为0代表从低地址值变为高地址。

12-14).

0xfd161: mov $0x8f, %eax

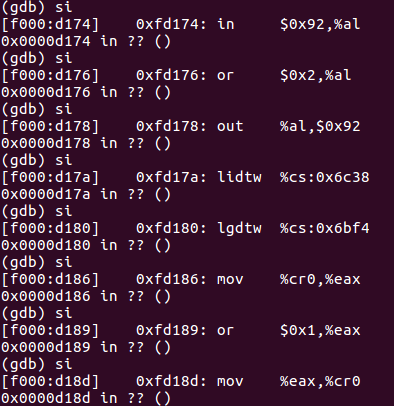
0xfd167: out %al, $0x70

0xfd169: in $0x71, %al

这两个操作是用于操作IO端口的。为了方便管理，80x86CPU采用IO端口单独编址的方式，即所有设备的端口都被命名到一个IO端口地址空间中。这个空间是独立于内存地址空间的。所以必须采用和访问内存的指令不一样的指令来访问端口。

in %al, PortAddress 向端口地址为PortAddress的端口写入值，值为al寄存器中的值。

out PortAddres,%al 把端口地址为PortAddress的端口中的值读入寄存器al中。

用al寄存器作为缓冲是规定，0x70端口和0x71端口是用于控制系统中一个叫做CMOS的设备，这个设备是一个低功耗的存储设备，它可以用于在计算机关闭时存储一些信息，它是由独立的电池供电的。 这个CMOS中可以控制跟PC相关的多个功能，其中最重要的就是时钟设备的 ，它还可以控制是否响应不可屏蔽中断NMI。

15-17).

0xfd16b: in $0x92, %al

0xfd16d: or $0x2, %al

0xfd16f: out %al, $0x92

把A20位使能，不清楚为啥，网上说是测试可用的内存空间。

18）.0xfd171: lidtw %cs:0x6ab8

加载中断向量表寄存器(IDTR)。这个指令会把从地址0xf6ab8起始的后面6个字节的数据读入到中断向量表寄存器(IDTR)中。

19）.0xfd177: lgdtw %cs:0x6a74

把从0xf6a74为起始地址处的6个字节的值加载到全局描述符表格寄存器中GDTR中。

20-22）.

0xfd17d: mov %cr0, %eax

0xfd180: or $0x1, %eax

0xfd184: mov %eax, %cr0

三个语句的操作明显是要把CR0寄存器的最低位(0bit)置1。CR0寄存器的0bit是PE位，启动保护位，当该位被置1，代表开启了保护模式。应该是现在实模式下工作，然后切换到保护模式。后面看不下去了，不会的东西越来越多，需要学习一个实模式和保护模式。

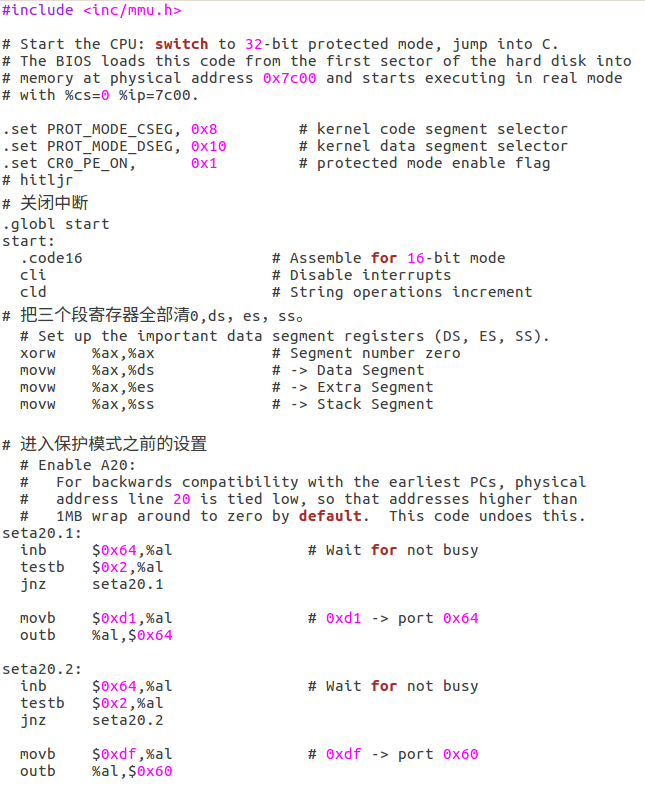
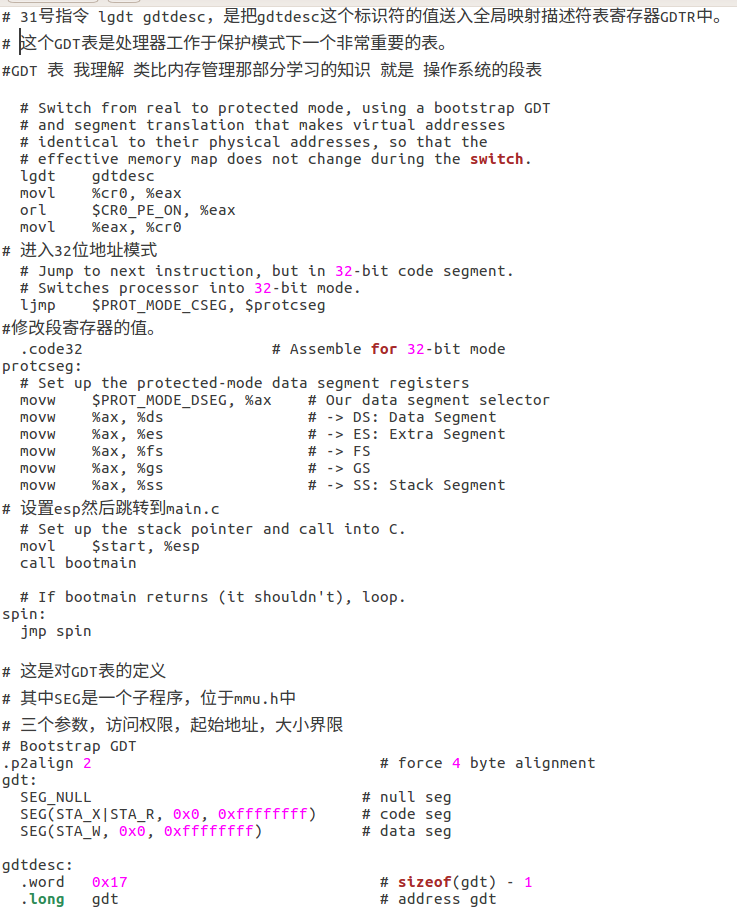
它最重要的功能是把操作系统从磁盘中导入内存，然后再把控制权转交给操作系统。

第二部分：引导程序

我们启动6.828,我们将采用传统的硬盘启动机制，这意味着我们的启动加载程序必须适合512字节。引导加载程序由boot/boot.s和boot/main.c 组成，我们需要仔细查看这两个源文件。

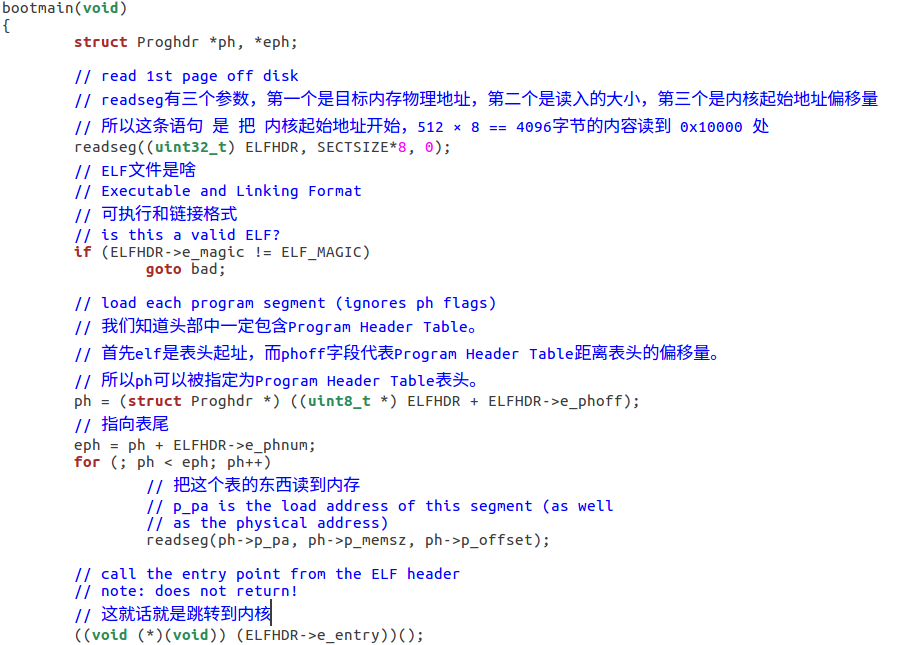
两个源文件的注释+代码：

Boot.s:



之后切换到32位保护模式，初始化栈,跳转到C代码。

将硬盘中的代码加载到物理地址0x7c00然后在实模式下执行。

关于main.c的注释如下。

磁盘布局：

存储在磁盘的第一个扇区

第二个扇区继续保存内核

内核必须采用ELF格式

引导程序需要完成两个功能：

1）.将处理器从实模式切换到32位保护模式，只有这样软件才能访问处理器物理地址空间1MB以上的内容。我们只需要了解，在保护模式下，将分段地址转换为物理的地址的方式不同，转换之后偏移量是32位而不是16位的。

2）.引导加载程序通过x86的特殊I\O指令发改问磁盘设备寄存器。

在你理解引导程序之后，需要查看文件boot.asm，这个文件是GNU编译引导程序生成的反汇编程序，可以通过这个反汇编程序，看物理内存，跟踪GDB。

并且可以在设置断点，b指令，C指令继续，SI一步一步执行，X/I指令查看内存中指令。

1. 练习三

学习GDB指令，然后设置断点，分步运行引导程序。

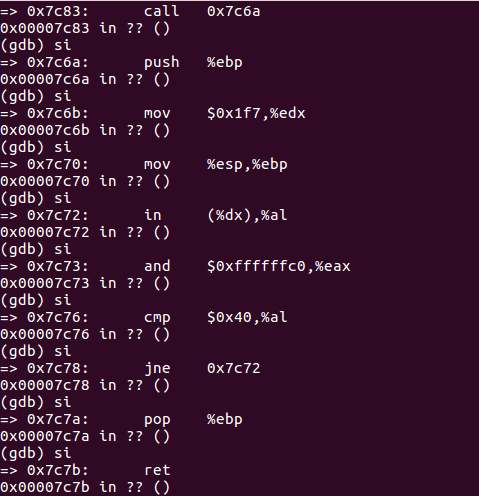
在地址0x7c00处设置断点，这是引导扇区将被加载的地方。

继续执行，直到该断点。

跟踪boot / boot.S中的代码，使用源代码和反汇编文件obj / boot / boot.asm来跟踪您的位置。还可以使用GDB中的x / i命令来分解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。跟踪到boot / main.c中的bootmain（）

然后进入readsect（）。

确定与readsect（）中的每个语句对应的精确汇编指令。

跟踪readsect（）的其余部分，并返回到bootmain（），并确定for循环的开始和结束，以便从磁盘读取内核的其余部分。找出循环结束时要运行的代码，在那里设置一个断点，

并继续到那个断点。然后逐步执行引导加载程序的其余部分。

回答问题：

问题1：处理器在什么时候开始执行32位代码？究竟什么原因导致从16位模式切换到32位模式？

在0x7c2d处的指令

ljmp $PROT\_MODE\_CSEG, $protcseg

跳转到了32位代码处，即从0x7c32处开始执行32位代码。在0x7c2a处开启cr0寄存器的PE位，从16位实模式转换到32位保护模式

问题2：什么是执行引导加载程序的最后一个指令，它刚加载的内核的第一条指令是什么？

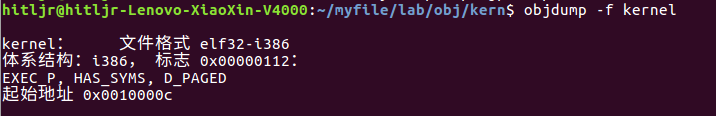
0x7d6b处的指令

call ×0x10018

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

其中ELF文件头的e\_entry字段的含义是这个可执行文件的第一条指令的虚拟地址。所以这句话的含义就是把控制权转移给操作系统内核。

问题3：内核的第一条指令在哪里？

 0X10000c/kern/entry.S

问题4：引导加载程序如何决定为了从磁盘获取整个内核而必须读取多少扇区？它在哪里找到这些信息？

每个段又有多少个扇区的信息位于操作系统文件中的Program Header Table中。存放在操作系统内核映像文件的ELF头部信息中。

关于/boot/boot.s和/boot/main.c,这两个文件一起组成了boot loader。

1. 练习4:复习C语言

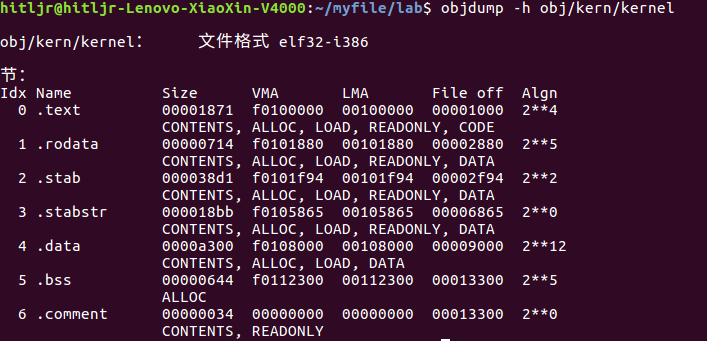
编译器把C转换成.o然后链接器把所有编译后的目标文件合并成一个单一二进制，ELF格式就是二进制文件，代表“可执行和可链接的格式”

在6.828中，你可以认为一个可以执行的ELF文件是由三大部分组成：一个是带有加载信息的文件头，然后紧跟着程序段表，然后紧跟着的就是几个程序段(program section)。其中每一个段都是一块连续的代码或者数据。它们在被运行时要首先被加载到内存中。boot loader的工作就是把它们加载到内存中。在每一个段中都有两个比较重要的字段，VMA(链接地址)，LMA(加载地址)。其中加载地址代表的就是这个段被加载到内存中后，它所在的物理地址。链接地址则指的是这个段希望被存放到的逻辑地址。.text:程序的可执行指令

.rodata：只读数据，例如由C编译器生成的ASCII字符串常量。

.data：数据部分保存程序的初始化数据，例如用初始化程序（如int x = 5;）声明的全局变量。

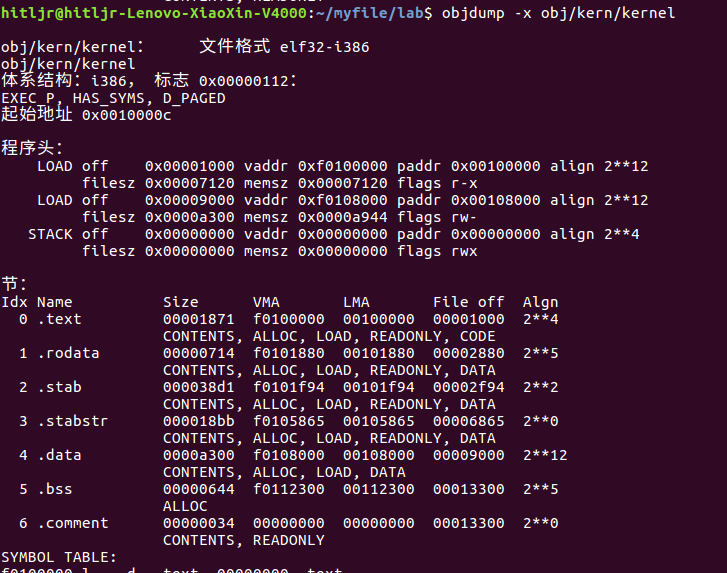
全局变量为0

 输入objdump -h obj/kern/kernel查看Program Headers Table

链接和加载地址是相同的

 例如我们查看.text部分

引导加载程序使用ELF程序头来决定如何加载部分。 程序标题指定了要加载到内存中的ELF对象的哪些部分以及每个应该占据的目标地址。

 输入 objdump -x obj/kern/kernel

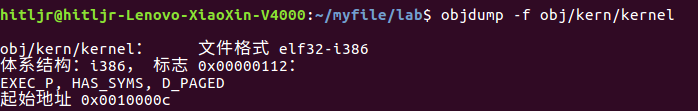
1. 练习5：

再次追踪，保证第一条指令错误，看看会发生什么

（测试完了别忘了改回去）链接地址可以理解为通过编译器链接器处理形成的可执行程序中指令的地址，即逻辑地址。加载地址则是可执行文件真正被装入内存后运行的地址，即物理地址。

输入：

objdump -f obj/kern/kernel

再次查看起始地址的信息会被改变。

1. 练习6：

在BIOS进入引导加载程序时检查0x00100000处的8个字的内存。

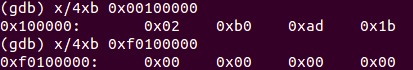
然后在引导加载程序进入内核时再次检查。

因为bootmain函数在最后会把内核的各个程序段送入到内存地址0x00100000处，所以这里现在存放的就是内核的某一个段的内容，由于程序入口地址是0x0010000C，正好位于这个段中，所以可以推测，这里面存放的应该是指令段，即.text段的内容。

第三部分：内核

现在，我们只需映射第一个4MB的物理内存，这足以让我们启动并运行。我们使用kern / entrypgdir.c中的手写静态初始化页面目录和页面表来完成此操作。现在，你不必了解这个工作的细节，只是它完成的效果。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boot / boot.S设置从线性地址到物理地址的身份映射，我们永远不会改变这一点）。一旦设置了CR0\_PG，内存引用就是由虚拟内存硬件转换为物理地址的虚拟地址。 entry\_pgdir将范围从0xf0000000到0xf0400000的虚拟地址转换为物理地址0x00000000到0x00400000，以及将虚拟地址0x00000000到0x00400000转换为物理地址0x00000000到0x00400000。

1. 练习7:

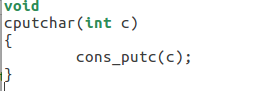
使用QEMU和GDB追踪JOS内核，并在movl％eax，％cr0处停止。 检查内存在0x00100000和0xf0100000。

现在，使用stepi GDB命令单步执行该指令。

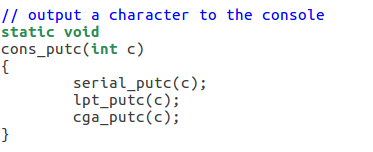
1. 练习8:

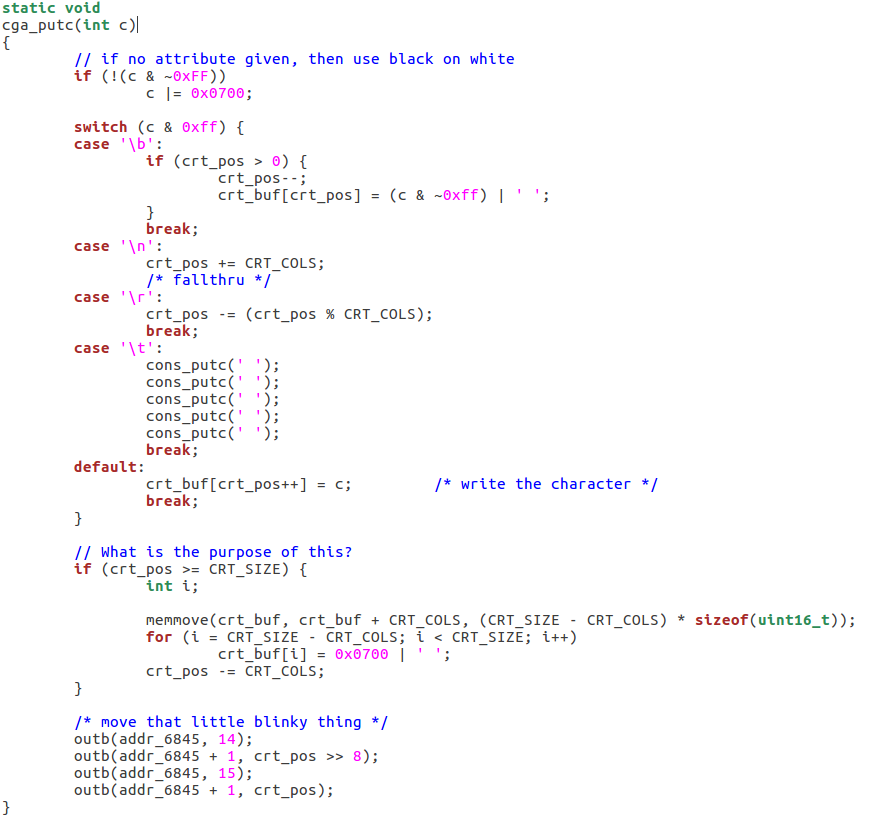
格式化打印到控制台

这一部分包括三个文件，kern\printf.c，\kern\console.c, \lib\printfmt.c

首先看一下\kern\console.c

接着深入看下去

所以serial\_putc子程序的功能是把一个字符输出给串口。lpt\_putc把这个字符输出给并口设备.最后一个函数把字符输出到cga设备上面，定义一个缓冲区。

具体来看cga\_putc

\lib\printfmt.c打印各种字符串的函数都在这，被printf，sprintf，fprintf函数使用。

这就是这个实验需要填写的部分。

Vprintfmt 有四个参数。

（void (\*putch)(int, void\*), void \*putdat, const char \*fmt, va\_list ap）

第一个参数：这类函数包含两个输入参数int, void\*，int参数代表一个要输出的字符的值。void\* 则代表要把这个字符输出的位置的地址。

第二个参数：这个参数就是输入的字符要存放在的内存地址的指针。

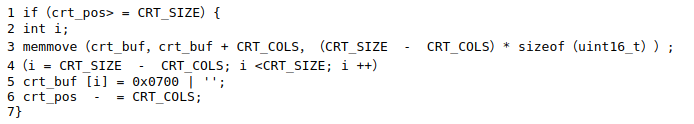
第三个参数：格式化字符串。

第四个参数：多个输入参数。

一个while循环处理，处理直接输出，处理格式化字符。

回答问题1. 解释一下printf.c和console.c两个之间的关系。console.c输出了哪些子函数？这些子函数是怎么被printf.c所利用的？

　在console.c中除了被static修饰符修饰的函数之外，都可以被外部所使用，其中被printf所使用的函数就是cputchar子函数。

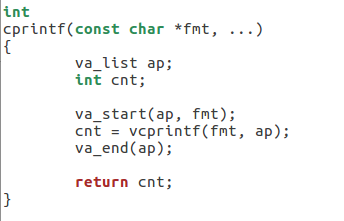
回答问题2.解释代码

crt\_buf:这是一个字符数组缓冲区，里面存放着要显示到屏幕上的字符

crt\_pos:这个表示当前最后一个字符显示在屏幕上的位置，光标位置？

当crt\_pos >= CRT\_SIZE，其中CRT\_SIZE = 80\*25，由于我们知道crt\_pos取值范围是0~(80\*25-1)，那么这个条件如果成立则说明现在在屏幕上输出的内容已经超过了一页。所以此时要把页面向上滚动一行，即把原来的1~79号行放到现在的0~78行上，然后把79号行换成一行空格（当然并非完全都是空格，0号字符上要显示你输入的字符int c）。所以memcpy操作就是把crt\_buf字符数组中1~79号行的内容复制到0~78号行的位置上。而紧接着的for循环则是把最后一行，79号行都变成空格。最后还要修改一下crt\_pos的值。

回答问题3.解释下面代码的参数

我们找到cprintf函数

这是cprintf的源代码，它调用了vcprintf，其中fmt格式字符串就是"x %d, y %x, z %d\n"，ap指向的是所有输入参数的集合。

回答问题4，运行下面代码。

我们会发现输出的是He110 World。

为什么会输出这样的值，首先看下第一个%x，指的是要按照16进制输出第一个参数，第一个参数的值是57616，它对应的16进制的表示形式为e110，所以前面就变成的He110。

然后看下一个%s，输出参数所指向的字符串。参数是&i，是变量i的地址，所以应该输出的是变量i所在地址处的字符串。

而在cprintf之前我们把i定义为一个int类型变量，所以现在我们要把它们进行拆分，按照一个字节一个字节来进行输出。由于x86是小端模式，代表字的最高位字节存放在最高位字节地址上。假设i变量的地址为0x00，那么i的4个字节的值存放在0x00，0x01，0x02，0x03四处。由于是小端存储，所以0x00处存放0x72('r')，0x01处存放0x6c('l')，0x02处存放0x64('d')，0x03处存放0x00('\0').

所以在cprintf将会从i的地址开始一个字节一个字节遍历，正好输出 "World"。

回答问题5：参数缺省

由于没有指定和数字，所以会随机输出

1. 练习9

1.判断一下操作系统内核是从哪条指令开始初始化它的堆栈空间的，以及这个堆栈坐落在内存的哪个地方？内核是如何给它的堆栈保留一块内存空间的？堆栈指针又是指向这块被保留的区域的哪一端的呢？

前面已经分析过boot.S和main.c文件的运行过程，这个文件中的代码是PC启动后，BIOS运行完成后，首先执行的两部分代码。但是它们并不属于操作系统的内核。当main.c文件中的bootmain函数运行到最后时，它执行的最后一条指令就是跳转到entry.S文件中的entry地址处。此时控制权已经被转交给了entry.S。

在跳转到entry之前，并没有对%esp，%ebp寄存器的内容进行修改，可见在bootmain中并没有初始化堆栈空间的语句。下面进入entry.S，在entry.S中我们可以看到它最后一条指令是要调用i386\_init()子程序。这个子程序位于init.c文件之中。在这个程序中已经开始对操作系统进行一些初始化工作，并且自重进入mointor函数。可见到i386\_init子程序时，内核的堆栈应该已经设置好了。所以设置内核堆栈的指令就应该是entry.S中位于 call i386\_init 指令之前的两条语句：

　movl $0x0,%ebp

　　movl $(bootstacktop),%esp

2.堆栈在内存的什么位置

所以用于堆栈的地址空间为 0xf0108000-0xf0110000，其中栈顶指针指向0xf0110000. 那么这个堆栈实际坐落在内存的 0x00108000-0x00110000物理地址空间中。

3.内核是如何给它的堆栈保留一块内存空间的？

其实就是通过刚刚分析的，在entry.S中的数据段里面声明一块大小为32Kb的空间作为堆栈使用。从而为内核保留了一块空间。

4.堆栈指针的指向

最高地址处

1. 练习10

为了能够更好的了解在x86上的C程序调用过程的细节，我们首先找到在obj/kern/kern.asm中test\_backtrace子程序的地址，设置断点，并且探讨一下在内核启动后，这个程序被调用时发生了什么。对于这个循环嵌套调用的程序test\_backtrace，它一共压入了多少信息到堆栈之中。并且它们都代表什么含义？

这四个操作将被用于存放调用这个子程序的父程序的栈帧信息，以及为当前子程序分配新的栈帧。在Exercise 1.9中，我们已经讨论过，entry.S文件中为整个内核设置了堆栈空间的地址范围，从0xf0108000-0xf0110000。由于堆栈是向下增长的，所以在运行init函数之前，esp寄存器的值就是0xf0110000，代表堆栈尚未使用。进入i386\_init函数后，如果要调用某个子程序，就会把 i386\_init 程序的栈帧信息压入到这个堆栈空间中。在 i386\_init 函数中，运行了子程序test\_backtrace(5)。当运行test\_backtrace(5)之前，esp寄存器ebp寄存器的值分别为如下：

esp ： 0xf010ffe0 ebp : 0xf010fff8

0xf010ffe0~0xf010fff8就是当前i386\_init子程序的栈帧，当计算机要调用test\_backtrace(5)程序时，首先call指令把i386\_init的返回地址压入堆栈中，所以esp变为0xf010ffdc，然后进入test\_backtrace(5)子程序。

子程序中第一句push %ebp，把i386\_init的ebp寄存器的值压入堆栈中，即地址0xf010ffd8处，此时esp的值变为0xf010ffd8。

然后 mov %esp, %ebp 把ebp的值更新为esp的值，0xf010ffd8。这个就是当前test子程序的ebp寄存器的值。即它的栈帧的高地址边界。

然后 push %ebx 把ebx寄存器的值压入堆栈，此时esp变为0xf010ffd4。因为%ebx寄存器可能被这个子程序所使用，所以必须把它之前的值保留。

然后 sub $0x14, %esp 把esp中的值减去0x14=20，esp的值变为0xf010ffc0。这就是给test子程序分配一个大小为20个存储单元的额外的栈帧空间，供它存储一些临时变量的值。

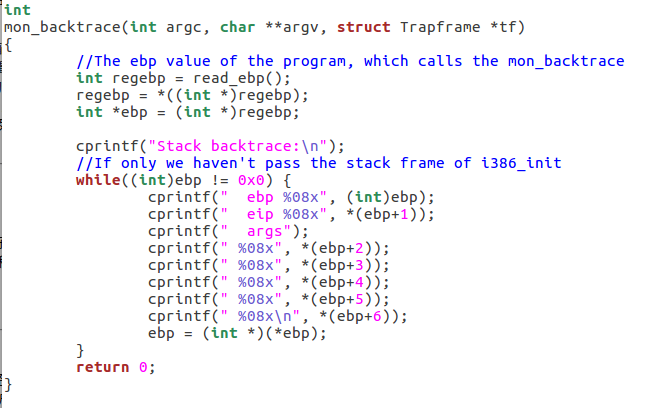
所以上述4条汇编指令，执行完成后，esp，ebp寄存器的值变化为

esp : 0xf010ffc0 ebp：0xf010ffd8

　对于任意一层调用，比如test\_backtrace(i)，它的esp和ebp的值假设分别为esp(i)和ebp(i)。那么在这个栈帧范围内主要存在这么几个重要的值：

　首先ebp(i)所指向的内存单元处存放着上一层程序的ebp寄存器的值，即ebp(i-1)。

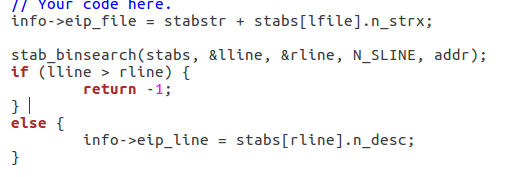
　另外在esp(i)所指向的内存单元处存放着对下一层子程序调用时传入的参数，即i+1

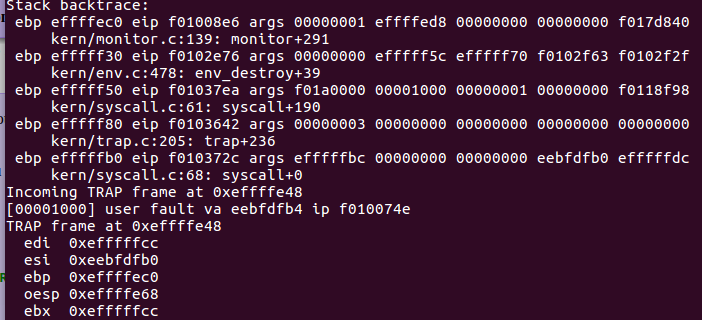
1. 练习11实现堆栈回溯函数

编写堆栈回溯函数：

这个子程序的功能就是要显示当前正在执行的程序的栈帧信息。包括当前的ebp寄存器的值，这个寄存器的值代表该子程序的栈帧的最高地址。eip则指的是这个子程序执行完成之后要返回调用它的子程序时，下一个要执行的指令地址。后面的值就是这个子程序接受的来自调用它的子程序传递给它的输入参数。

1. 练习12

但是，在实践中，经常想知道这些地址对应的函数名称。例如，可能想知道哪些函数可能包含导致您的内核崩溃的错误。为了实现这个功能，我们提供了在符号表中debuginfo\_eip()查找eip的函数 ，并返回该地址的调试信息。这个函数在kern / kdebug.c中定义。

查看信息

### 2.实验结果

## Lab 2: Memory Management

### 实验目的

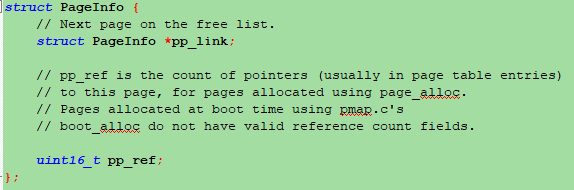
操作系统必须要追踪记录哪些内存区域是空闲的，哪些是被占用的。JOS内核是以页(page)为最小粒度来管理内存的，它使用MMU来映射，保护每一块被分配出去的内存。在这里你要具体编写一下物理内存页的分配子函数。它利用一个结构体PageInfo的链表来记录哪些页是空闲的，链表中每一个结点对应一个物理页。

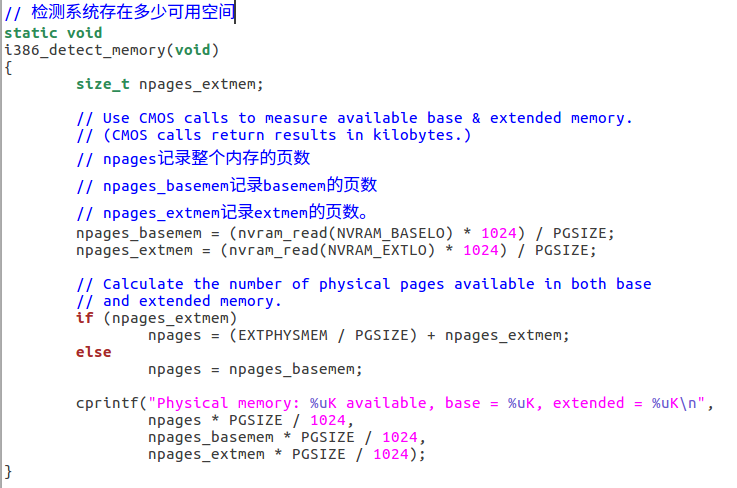
### 实验内容（附过程截图）

1. 练习1：

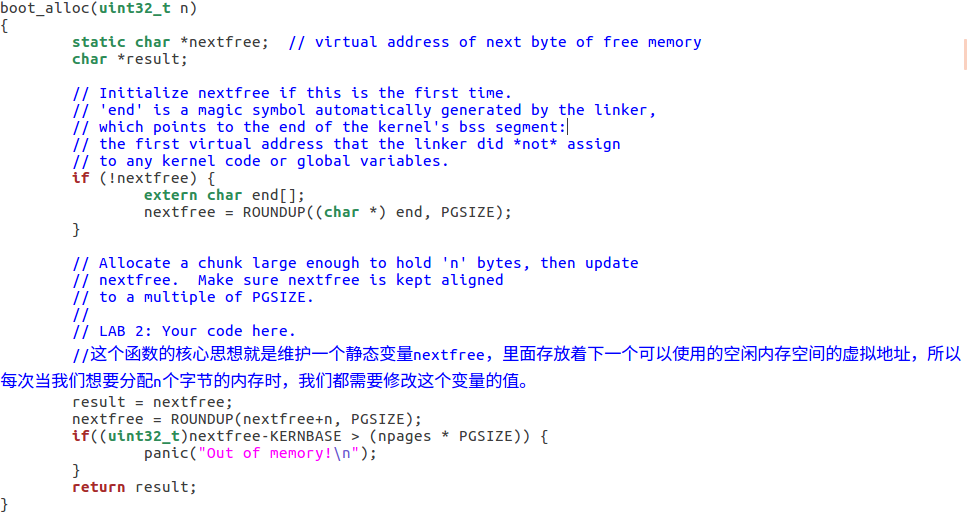
在文件 kern/pmap.c 中，你必须要完成以下几个子函数的代码boot\_alloc(); mem\_init(); page\_init(); page\_alloc();page\_free();

这是pageinfo的结构



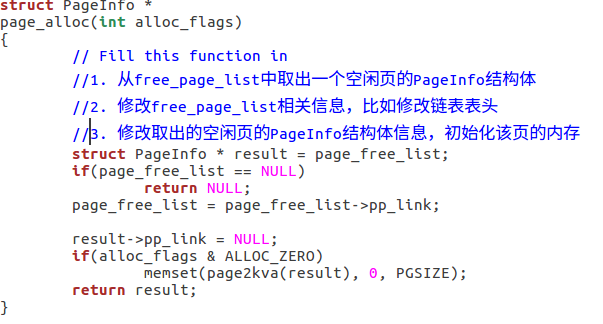
 先来看mem\_init()的代码。

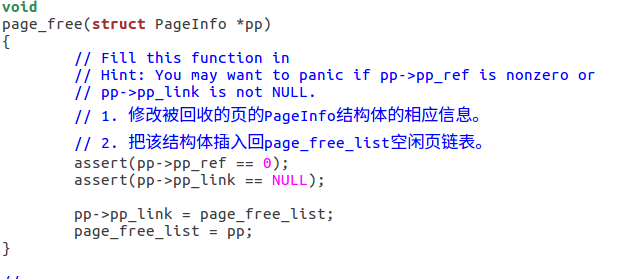
这是 i386\_detect\_memory 子函数的代码及注释。

下面到了第一个需要添加的部分。

Page\_init()需要补全的第二个函数

接下来就是page\_alloc() 和page\_free()两个函数。

先实现page\_alloc()函数，通过注释我们可以知道这个函数的功能就是分配一个物理页。而函数的返回值就是这个物理页所对应的PageInfo结构体。

实现page\_free()方法，根据注释可知，这个方法的功能就是把一个页的PageInfo结构体再返回给page\_free\_list空闲页链表，代表回收了这个页。

一个虚拟地址(Virtual Address)是由两部分组成，一个是段选择子(segment selector)，另一个是段内偏移(segment offset)。一个线性地址(Linear Address)指的是通过段地址转换机构把虚拟地址进行转换之后得到的地址。一个物理地址(Physical Addresses)是分页地址转换机构把线性地址进行转换之后得到的真实的内存地址，这个地址将会最终送到你的内存芯片的地址总线上。

xp/Nx paddr – 查看paddr物理地址处开始的，N个字的16进制的表示结果。

info registers – 展示所有内部寄存器的状态。

info mem – 展示所有已经被页表映射的虚拟地址空间，以及它们的访问优先级。

info pg – 展示当前页表的结构。

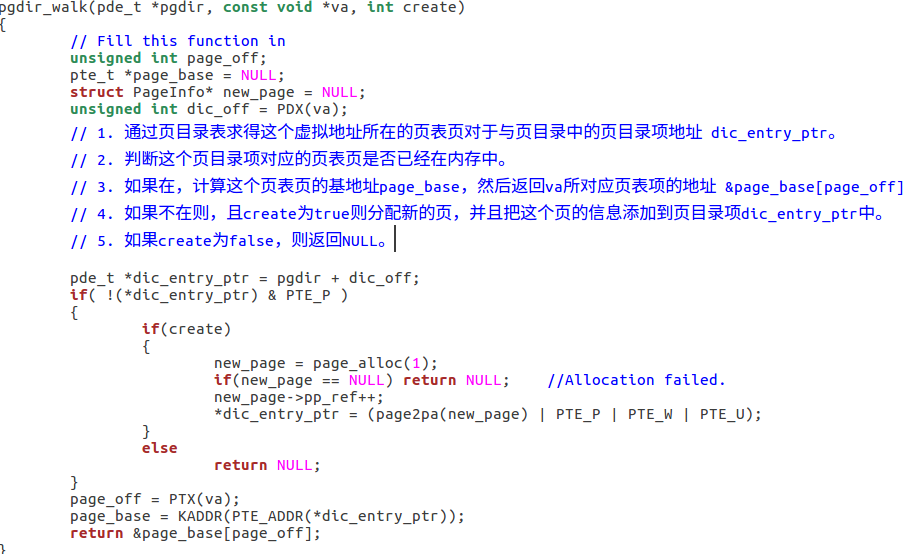
1. 练习2：

参阅“ 英特尔80386参考手册”的第5章和第6章 ，如果还没有这样做的话。仔细阅读关于页面翻译和页面保护的章节（5.2和6.4）。浏览有关细分的部分; 而JOS使用分页硬件进行虚拟内存和保护时，在x86上不能禁用段转换和基于段的保护，因此需要对其进行基本的了解。

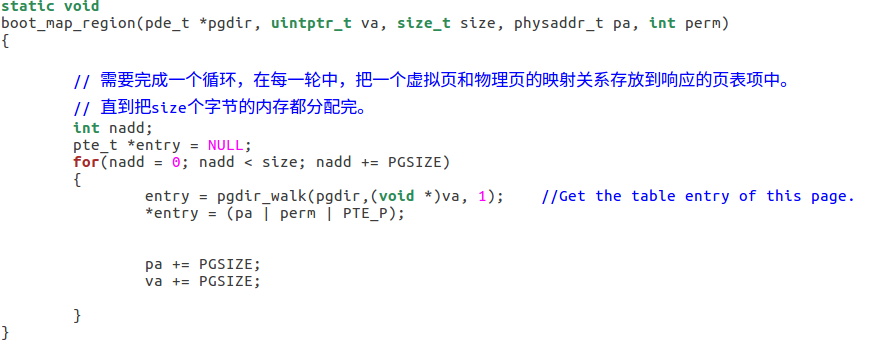
1. 了解QEMU的一些指令，查看内存。
2. 练习4：

完成pgdir\_walk()boot\_map\_region()page\_lookup() page\_remove()page\_insert()

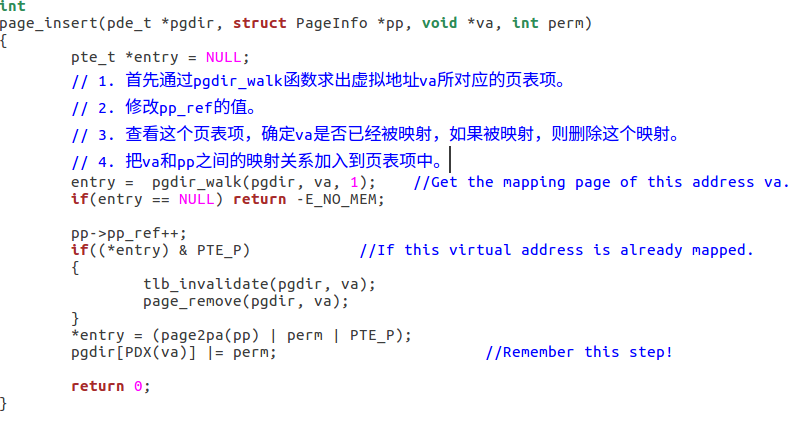
函数给定一个页目录表指针 pgdir ，该函数应该返回线性地址va所对应的页表项指针。

代码截图如下：

boot\_map\_region函数，把虚拟地址空间范围[va, va+size)映射到物理空间[pa, pa+size)的映射关系加入到页表pgdir中。这个函数主要的目的是为了设置虚拟地址UTOP之上的地址范围，这一部分的地址映射是静态的，在操作系统的运行过程中不会改变，所以这个页的PageInfo结构体中的pp\_ref域的值不会发生改变。

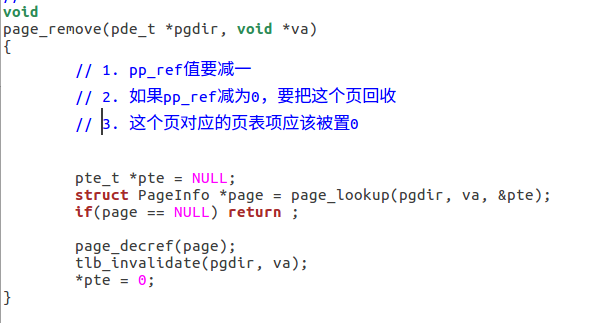
 代码截图

page\_insert()

功能上是完成：把一个物理内存中页pp与虚拟地址va建立映射关系。

pp->pp\_ref++这条语句，一定要放在page\_remove之前。

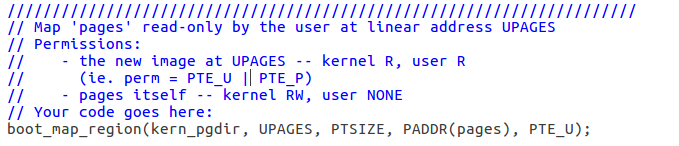
接下来继续完成page\_lookup()函数，返回虚拟地址va所映射的物理页的PageInfo结构体的指针，如果pte\_store参数不为0，则把这个物理页的页表项地址存放在pte\_store中。

page\_remove函数，功能就是把虚拟地址va和物理页的映射关系删除。

由于内核和用户进程只能访问各自的地址空间，所以我们必须在x86页表中使用访问权限位(Permission Bits)来使用户进程的代码只能访问用户地址空间，而不是内核地址空间。否则用户代码中的一些错误可能会覆写内核中的数据，最终导致内核的崩溃。处在用户地址空间中的代码不能访问高于ULIM的地址空间，但是内核可以读写这部分空间。而内核和用户对于地址范围[UTOP, ULIM]有着相同的访问权限，那就是可以读取但是不可以写入。这一个部分的地址空间通常被用于把一些只读的内核数据结构暴露给用户地址空间的代码。在UTOP之下的地址范围是给用户进程使用的，用户进程可以访问，修改这部分地址空间的内容。

1. 练习5：

剩下的工作就是要完善mem\_init()函数，现在要完善的功能就是把关于操作系统的一些重要的地址范围映射到现在的新页目录项上kern\_pgdir上。这里我们可以利用前面定义过的boot\_map\_region函数。

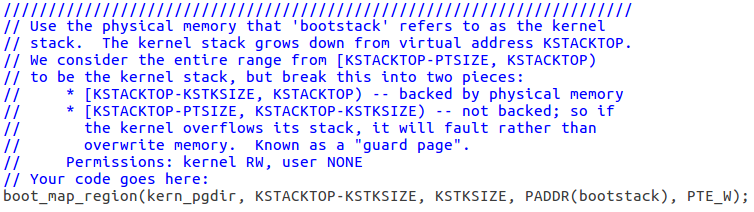
1. 首先我们要映射的范围是把pages数组映射到线性地址UPAGES，大小为一个PTSIZE。

其中perm变量之所以设置为PTE\_U，是因为这部分空间是kernel space和user space中的代码都能访问的，所以要设置PTE\_U。

1. 然后映射内核的堆栈区域，把由bootstack变量所标记的物理地址范围映射给内核的堆栈。内核堆栈的虚拟地址范围是[KSTACKTOP-PTSIZE, KSTACKTOP)，不过要把这个范围划分成两部分：

　\* [KSTACKTOP-KSTKSIZE, KSTACKTOP) 这部分映射关系加入的页表中。

\*[KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) 这部分不进行映射。

对这部分地址的访问权限是，kernel space 可以读写，user space 无权访问。：

1. 最后映射整个操作系统内核，虚拟地址范围是[KERNBASE, 2^32]，物理地址范围是[0，2^32 - KERNBASE]。

访问权限是，kernel space 可以读写，user space 无权访问。

1. 问题简单：

问题2：到目前为止页目录表中已经包含多少有效页目录项？他们都映射到哪里？

3BD号页目录项，指向的是kern\_pgdir

3BC号页目录项，指向的是pages数组

3BF号页目录项，指向的是bootstack

3C0~3FF号页目录项，指向的是kernel

问题3：如果我们把kernel和user environment放在一个相同的地址空间中。为什么用户程序不同读取，写入内核的内存空间？用什么机制保护内核的地址范围。

用户程序不能去随意修改内核中的代码，数据，否则可能会破坏内核，造成程序崩溃。

正常的操作系统通常采用两个部件来完成对内核地址的保护，一个是通过段机制来实现的，但是JOS中的分段功能并没有实现。二就是通过分页机制来实现，通过把页表项中的 Supervisor/User位置0，那么用户态的代码就不能访问内存中的这个页。

问题4： 这个操作系统的可以支持的最大数量的物理内存是多大？

由于这个操作系统利用一个大小为4MB的空间UPAGES来存放所有的页的PageInfo结构体信息，每个结构体的大小为8B，所以一共可以存放512K个PageInfo结构体，所以一共可以出现512K个物理页，每个物理页大小为4KB，自然总的物理内存占2GB。

问题5：如果现在的物理内存页达到最大个数，那么管理这些内存所需要的额外空间开销有多少？

首先需要存放所有的PageInfo，需要4MB，需要存放页目录表，kern\_pgdir，4KB，还需要存放当前的页表，大小为2MB。所以总的开销就是6MB + 4KB。

问题6：回顾entry.S文件中，当分页机制开启时，寄存器EIP的值仍旧是一个小的值。在哪个位置代码才开始运行在高于KERNBASE的虚拟地址空间中的？当程序位于开启分页之后到运行在KERNBASE之上这之间的时候，EIP的值是小的值，怎么保证可以把这个值转换为真实物理地址的？

在entry.S文件中有一个指令 jmp \*%eax，这个指令要完成跳转，就会重新设置EIP的值，把它设置为寄存器eax中的值，而这个值是大于KERNBASE的，所以就完成了EIP从小的值到大于KERNBASE的值的转换。在entry\_pgdir这个页表中，也把虚拟地址空间[0, 4MB)映射到物理地址空间[0, 4MB)上，所以当访问位于[0, 4MB)之间的虚拟地址时，可以把它们转换为物理地址。

由于内核和用户进程只能访问各自的地址空间，所以我们必须在x86页表中使用访问权限位(Permission Bits)来使用户进程的代码只能访问用户地址空间，而不是内核地址空间。否则用户代码中的一些错误可能会覆写内核中的数据，最终导致内核的崩溃。

处在用户地址空间中的代码不能访问高于ULIM的地址空间，但是内核可以读写这部分空间。而内核和用户对于地址范围[UTOP, ULIM]有着相同的访问权限，那就是可以读取但是不可以写入。这一个部分的地址空间通常被用于把一些只读的内核数据结构暴露给用户地址空间的代码。在UTOP之下的地址范围是给用户进程使用的，用户进程可以访问，修改这部分地址空间的内容。

### 实验结果

## Lab 3: User Environments

### 实验目的

在这个实验中，我们将实现操作系统的一些基本功能，来实现用户环境下的进程的正常运行。你将会加强JOS内核的功能，为它增添一些重要的数据结构，用来记录用户进程环境的一些信息；创建一个单一的用户环境，并且加载一个程序运行它。你也可以让JOS内核能够完成用户环境所作出的任何系统调用，以及处理用户环境产生的各种异常。

### 实验内容（附过程截图）

这个实验 多了很多代码

|  |  |  |
| --- | --- | --- |
| INC / | env.h | 用户模式环境的公共定义 |
|  | trap.h | 陷阱处理的公共定义 |
|  | syscall.h | 从用户环境到内核的系统调用的公共定义 |
|  | lib.h | 用户模式支持库的公共定义 |
| kern/ | env.h | 用户模式环境的内核专用定义 |
|  | env.c | 内核代码实现用户模式环境 |
|  | trap.h | 内核专用陷阱处理定义 |
|  | trap.c | 陷阱处理代码 |
|  | trapentry.S | 汇编语言陷阱处理程序入口点 |
|  | syscall.h | 系统调用处理的内核专用定义 |
|  | syscall.c | 系统调用实现代码 |
| Lib/ | Makefrag | Makefile片段构建用户模式库 obj / lib / libjos.a |
|  | entry.S中 | 用户环境的汇编语言入口 |
|  | libmain.c | 来自entry.S的用户模式库设置代码 |
|  | syscall.c | 用户模式系统调用存根函数 |
|  | console.c中 | putchar和getchar的用户模式实现，控制台I / O |
|  | exit.c中 | 用户模式执行的退出 |
|  | panic.c | 用户模式执行 panic |
| user/ | \* | 各种测试程序来检查内核实验3代码 |

操作系统维护了三个重要的全局变量

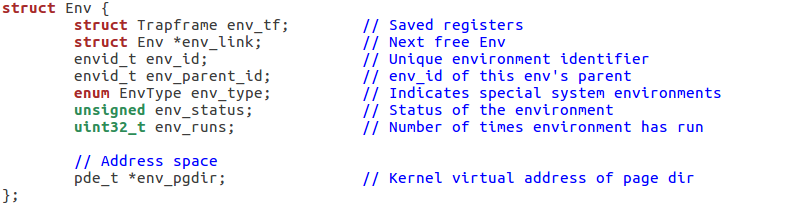
　 struct Env \*envs = NULL;//所有的 Env 结构体

struct Env \*curenv = NULL; //目前正在运行的用户环境

static struct Env \*env\_free\_list;

//还没有被使用的 Env 结构体链表

内核刚启动的时候，curenv的值为NULL

具体Env 结构体定义 env\_tf:

　　这个类型的结构体在inc/trap.h文件中被定义，里面存放着当用户环境暂停运行时，所有重要寄存器的值。内核也会在系统从用户态切换到内核态时保存这些值，这样的话用户环境可以在之后被恢复，继续执行。

　　env\_link:

　　这个指针指向在env\_free\_list中，该结构体的后一个free的Env结构体。当然前提是这个结构体还没有被分配给任意一个用户环境时，该域才有用。

　　env\_id:

　　这个值可以唯一的确定使用这个结构体的用户环境是什么。当这个用户环境终止，内核会把这个结构体分配给另外一个不同的环境，这个新的环境会有不同的env\_id值。

　　env\_parent\_id:

创建这个用户环境的父用户环境的env\_id

env\_type:

用于区别出来某个特定的用户环境。对于大多数环境来说，它的值都是 ENV\_TYPE\_USER.

env\_status:

这个变量存放以下可能的值

*ENV\_FREE: 代表这个结构体是不活跃的，应该在链表env\_free\_list中。*

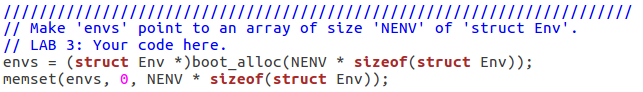
*ENV\_RUNNABLE: 代表这个结构体对应的用户环境已经就绪，等待被分配处理机。*

*ENV\_RUNNING: 代表这个结构体对应的用户环境正在运行。*

*ENV\_NOT\_RUNNABLE: 代表这个结构体所代表的是一个活跃的用户环境，但是它不能被调度运行，因为它在等待其他环境传递给它的消息。*

*ENV\_DYING: 代表这个结构体对应的是一个僵尸环境。一个僵尸环境在下一次陷入内核时会被释放回收。*

env\_pgdir: 这个变量存放着这个环境的页目录的虚拟地址

1. 练习1

现在你需要进一步去修改mem\_init()函数，来分配一个Env结构体数组，叫做envs。

这个实验在做了lab2的实验之后就很好做了。

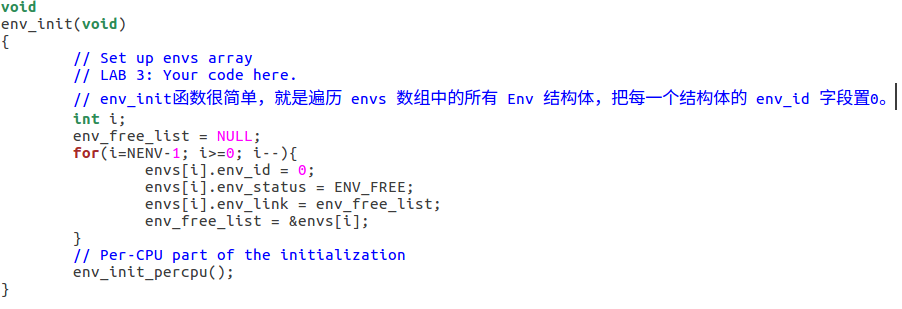
我们现在需要来运行一个用户环境，我们需要把内核设置成能够加载内核中的静态二进制文件。

1. 练习2

region\_alloc(): 为用户环境分配物理地址空间

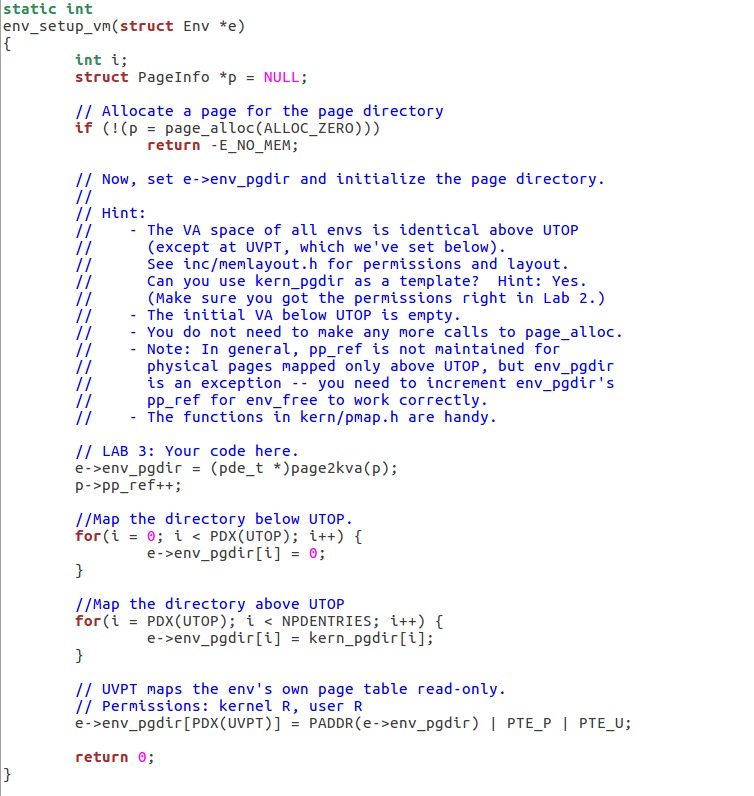
　　　load\_icode(): 分析一个ELF文件，类似于boot loader做的那样，我们可以把它的内容加载到用户环境下。

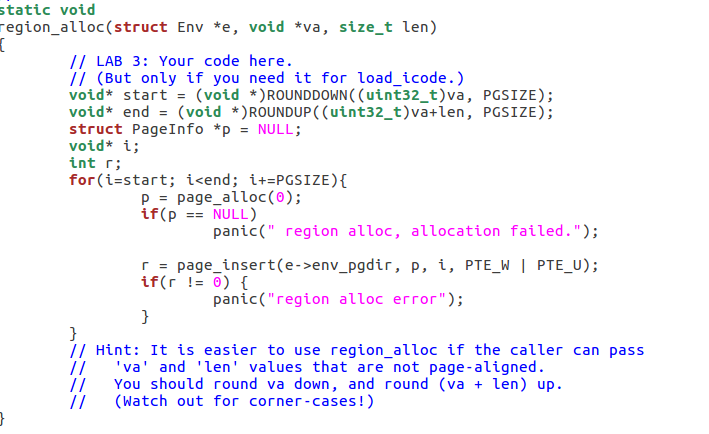
　　　env\_create(): 利用env\_alloc函数和load\_icode函数，加载一个ELF文件到用户环境中

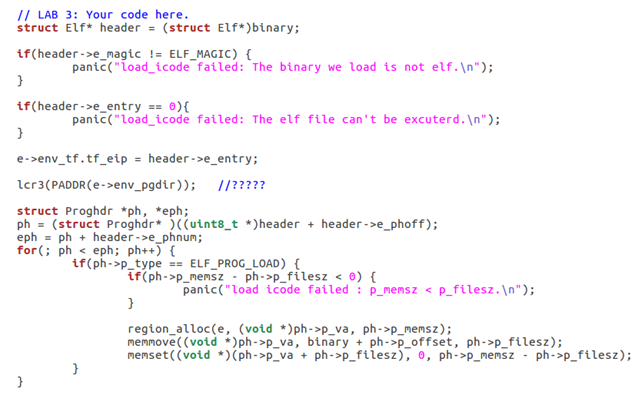
　　　env\_run(): 在用户模式下，开始运行一个用户环境。

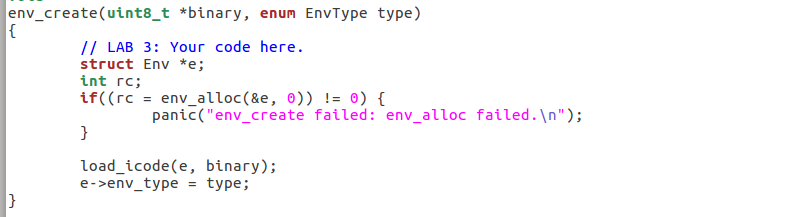
这是env\_init():初始化所有的在envs数组中的 Env结构体，并把它们加入到 env\_free\_list中。 还要调用 env\_init\_percpu，这个函数要配置段式内存管理系统，让它所管理的段，可能具有两种访问优先级其中的一种，一个是内核运行时的0优先级，以及用户运行时的3优先级。

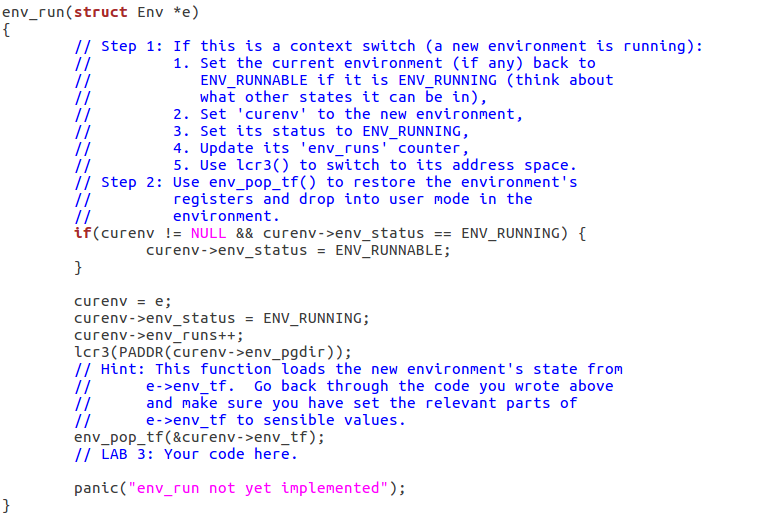
　env\_setup\_vm(): 为一个新的用户环境分配一个页目录表，并且初始化这个用户环境的地址空间中的和内核相关的部分。

region\_alloc 为用户环境分配物理空间，这里注意我们要先把起始地址和终止地址进行页对齐，对其之后我们就可以以页为单位，为其一个页一个页的分配内存，并且修改页目录表和页表。

load\_icode 功能是为每一个用户进程设置它的初始代码区，堆栈以及处理器标识位。每个用户程序都是ELF文件，所以我们要解析该ELF文件。



env\_create 是利用env\_alloc函数和load\_icode函数，加载一个ELF文件到用户环境中。

env\_run 是真正开始运行一个用户环境

用户环境的代码被调用前，操作系统一共按顺序执行了以下几个函数：

　\* start (kern/entry.S)

　\* i386\_init (kern/init.c)

cons\_init

mem\_init

env\_init

trap\_init （目前还未实现）

env\_create

env\_run

env\_pop\_tf

显然用户级线程是不可以调用系统内存的，因为现在系统无法从用户态切换到内核态。所以你需要实现一个基本的异常/系统调用处理机制，使得内核可以从用户态转换为内核态。

1. 中断向量表：

　　　处理器保证中断和异常只能够引起内核进入到一些特定的，被事先定义好的程序入口点，而不是由触发中断的程序来决定中断程序入口点。

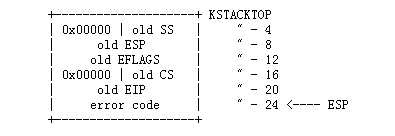
　　　X86允许多达256个不同的中断和异常，每一个都配备一个独一无二的中断向量。一个向量指的就是0到255中的一个数。一个中断向量的值是根据中断源来决定的：不同设备，错误条件，以及对内核的请求都会产生出不同的中断和中断向量的组合。CPU将使用这个向量作为这个中断在中断向量表中的索引，这个表是由内核设置的，放在内核空间中，和GDT很像。通过这个表中的任意一个表项，处理器可以知道：

　　　\*需要加载到EIP寄存器中的值，这个值指向了处理这个中断的中断处理程序的位置。

　　　\*需要加载到CS寄存器中的值，里面还包含了这个中断处理程序的运行特权级。（即这个程序是在用户态还是内核态下运行。）

　　2. 任务状态段

　　　处理器还需要一个地方来存放，当异常/中断发生时，处理器的状态，比如EIP和CS寄存器的值。这样的话，中断处理程序一会可以重新返回到原来的程序中。这段内存自然也要保护起来，不能被用户态的程序所篡改。

　　　正因为如此，当一个x86处理器要处理一个中断，异常并且使运行特权级从用户态转为内核态时，它也会把它的堆栈切换到内核空间中。一个叫做 “任务状态段（TSS）”的数据结构将会详细记录这个堆栈所在的段的段描述符和地址。处理器会把SS，ESP，EFLAGS，CS，EIP以及一个可选错误码等等这些值压入到这个堆栈上。然后加载中断处理程序的CS，EIP值，并且设置ESP，SS寄存器指向新的堆栈。

　　　　尽管TSS非常大，并且还有很多其他的功能，但是JOS仅仅使用它来定义处理器从用户态转向内核态所采用的内核堆栈，由于JOS中的内核态指的就是特权级0，所以处理器用TSS中的ESP0，SS0字段来指明这个内核堆栈的位置，大小。

扩展0-31号中断

1. trap\_init() 先将所有中断处理函数的起始地址放到中断向量表IDT中。

　　2. 当中断发生时，不管是外部中断还是内部中断，处理器捕捉到该中断，进入核心态，根据中断向量去查询中断向量表，找到对应的表项

　　3. 保存被中断的程序的上下文到内核堆栈中，调用这个表项中指明的中断处理函数。

　　4. 执行中断处理函数。

5. 执行完成后，恢复被中断的进程的上下文，返回用户态，继续运行这个进程。

1. 阅读 第9章异常和中断 的 80386程序员手册 （或第5章 IA-32开发者手册）
2. 练习四：

编辑一下trapentry.S 和 trap.c 文件，并且实现上面所说的功能。宏定义 TRAPHANDLER 和 TRAPHANDLER\_NOEC 会对你有帮助。你将会在 trapentry.S文件中为在inc/trap.h文件中的每一个trap加入一个入口指， 你也将会提供\_alttraps的值。

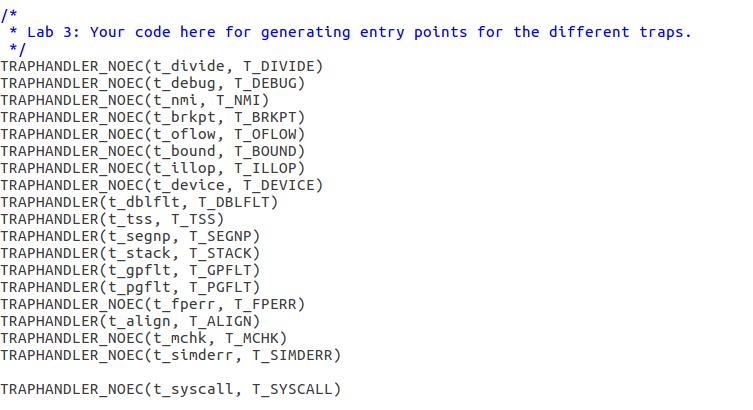
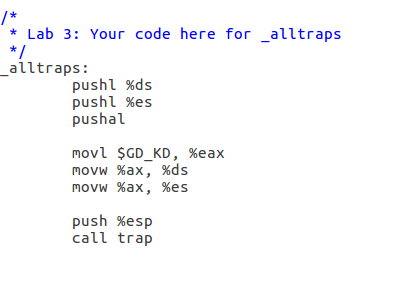
需要修改trap\_init()函数来初始化idt表，使表中每一项指向定义在trapentry.S中的入口指针，SETGATE宏定义在这里用得上。

你所实现的 \_alltraps 应该：

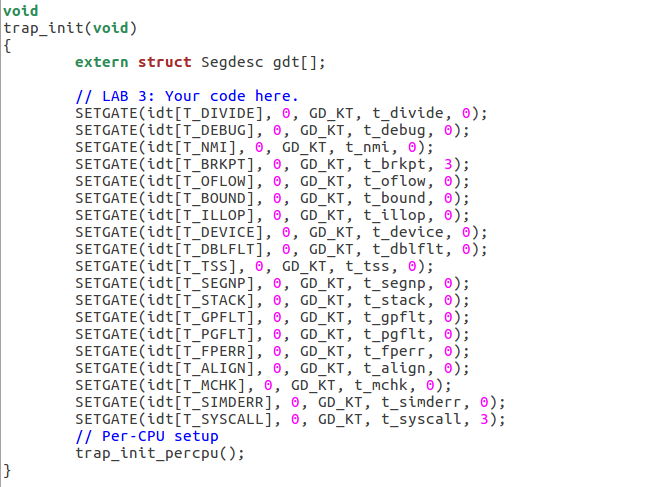
1. 把值压入堆栈使堆栈看起来像一个结构体 Trapframe

2. 加载 GD\_KD 的值到 %ds, %es寄存器中

3. 把%esp的值压入，并且传递一个指向Trapframe的指针到trap()函数中。

4. 调用trap trapentry.S

对于trap\_init如下

问答1

不同的中断或者异常当然需要不同的中断处理函数，因为不同的异常/中断可能需要不同的处理方式，比如有些异常是代表指令有错误，则不会返回被中断的命令。而有些中断可能只是为了处理外部IO事件，此时执行完中断函数还要返回到被中断的程序中继续运行。

问答2

因为当前的系统正在运行在用户态下，特权级为3，而INT指令为系统指令，特权级为0。特权级为3的程序不能直接调用特权级为0的程序，会引发一个General Protection Exception，即trap 13。

1. 练习5

修改一下 trap\_dispatch 函数，使系统能够把缺页异常引导到 page\_fault\_handler() 上执行。

根据 trapentry.S 文件中的 TRAPHANDLER 函数可知，这个函数会把当前中断的中断码压入堆栈中，再根据 inc/trap.h 文件中的 Trapframe 结构体我们可以知道，Trapframe 中的 tf\_trapno 成员代表这个中断的中断码。所以在 trap\_dispatch 函数中我们需要根据输入的 Trapframe 指针 tf 中的 tf\_trapno 成员来判断到来的中断是什么中断，这里我们需要判断是否是缺页中断，如果是则执行 page\_fault\_handler 函数。

断点异常，异常号为3，这个异常可以让调试器能够给程序加上断点。加断点的基本原理就是把要加断点的语句用一个 INT3 指令替换，执行到INT3时，会触发软中断。在JOS中，我们将通过把这个异常转换成一个伪系统调用，这样的话任何用户环境都可以使用这个伪系统调用来触发JOS kernel monitor。

系统调用，用户程序会要求内核帮助它完成系统调用。当用户程序触发系统调用，系统进入内核态。处理器和操作系统将保存该用户程序当前的上下文状态，然后由内核将执行正确的代码完成系统调用，然后回到用户程序继续执行。而用户程序到底是如何得到操作系统的注意，以及它如何说明它希望操作系统做什么事情的方法是有很多不同的实现方式的。



1. 练习6:

这个练习和上一个基本类似

断点异常，异常号为3，这个异常可以让调试器能够给程序加上断点。加断点的基本原理就是把要加断点的语句用一个 INT3 指令替换，执行到INT3时，会触发软中断。在JOS中，我们将通过把这个异常转换成一个伪系统调用，这样的话任何用户环境都可以使用这个伪系统调用来触发JOS kernel monitor。

问答：在上面的break point exception测试程序中，如果你在设置IDT时，对break point exception采用不同的方式进行设置，可能会产生触发不同的异常，有可能是break point exception，有可能是 general protection exception。这是为什么？你应该怎么做才能得到一个我们想要的breakpoint exception，而不是general protection exception？

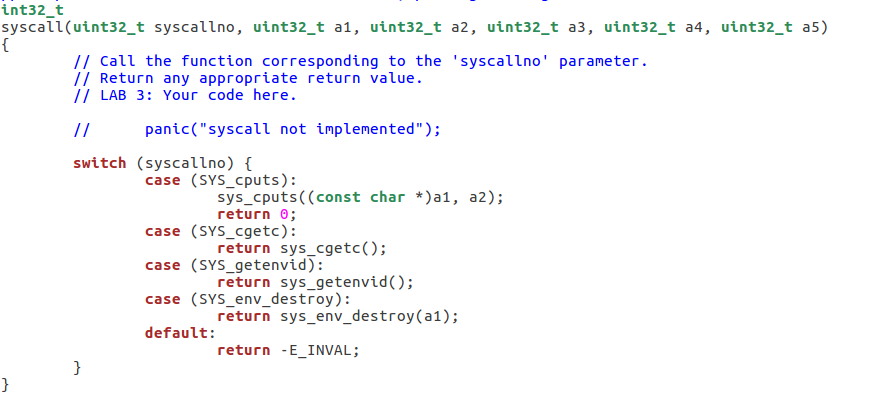
答：通过实验发现出现这个现象的问题就是在设置IDT表中的breakpoint exception的表项时，如果我们把表项中的DPL字段设置为3，则会触发break point exception，如果设置为0，则会触发general protection exception。DPL字段代表的含义是段描述符优先级（Descriptor Privileged Level），如果我们想要当前执行的程序能够跳转到这个描述符所指向的程序哪里继续执行的话，有个要求，就是要求当前运行程序的CPL，RPL的最大值需要小于等于DPL，否则就会出现优先级低的代码试图去访问优先级高的代码的情况，就会触发general protection exception。那么我们的测试程序首先运行于用户态，它的CPL为3，当异常发生时，它希望去执行 int 3指令，这是一个系统级别的指令，用户态命令的CPL一定大于 int 3 的DPL，所以就会触发general protection exception，但是如果把IDT这个表项的DPL设置为3时，就不会出现这样的现象了，这时如果再出现异常，肯定是因为我们还没有编写处理break point exception的程序所引起的，所以是break point exception。

1. 练习7

给中断向量T\_SYSCALL编写一个中断处理函数。你需要去编辑kern/trapentry.S和kern/trap.c中的trap\_init()函数。你也需要去修改trap\_dispatch()函数，使他能够通过调用syscall()（在kern/syscall.c中定义的）函数处理系统调用中断。最终你需要去实现kern/syscall.c中的syscall()函数。确保这个函数会在系统调用号为非法值时返回-E\_INVAL。你应该充分理解lib/syscall.c文件。我们要处理在inc/syscall.h文件中定义的所有系统调用。

我们需要了解一下系统调用的整个流程，如果现在运行的是内核态的程序的话，此时调用了一个系统调用，比如 sys\_cputs 函数时，此时不会触发中断，那么系统会直接执行定义在 lib/syscall.c 文件中的 sys\_cputs，我们可以看一下这个文件，可以发现这个文件中定义了几个比较常用的系统调用，包括 sys\_cputs, sys\_cgetc 等等。我们还会发现他们都是统一调用一个 syscall 函数，通过这个函数的代码发现其实它是执行了一个汇编指令。所以最终是这个函数完成了系统调用。以上是运行在内核态下的程序，调用系统调用时的流程。但是如果是用户态程序呢？这个练习就是让我们编写程序使我们的用户程序在调用系统调用时，最终也能经过一系列的处理最终去执行 lib/syscall.c 中的 syscall 指令。让我们看一下这个过程，当用户程序中要调用系统调用时，比如 sys\_cputs，从它的汇编代码中我们会发现，它会执行一个 int $0x30 指令，这个指令就是软件中断指令，这个中断的中断号就是 0x30，即 T\_SYSCALL，所以题目中让我们首先为这个中断号编写一个中断处理函数，我们首先就要在 kern/trapentry.S 文件中为它声明它的中断处理函数，即TRAPHANDLER\_NOEC，就像我们为其他中断号所做的那样。

所以我们可以假象一下，是不是 kern/syscall.c 中的 syscall 就是一个外壳函数，它的存在就是为了能够调用 lib/syscall 的呢？ 所以我们按照这个思路继续进行下去，我们再继续观察 kern/syscall.c 中的其他函数，会惊人的发现，kern/syscall.c 中的所有函数居然和 lib/syscall.c 中的所有函数都是一样的！！比如 在这两个文件中都有 sys\_cputs 函数，但是我们仔细观察可以发现这两个同名的函数，实现方式却不一样。

 其实系统调用，syscall函数需要处理各种函数的信息

1. 练习8：

用户程序真正开始运行的地方是在lib/entry.S文件中。该文件中，首先会进行一些设置，然后就会调用lib/libmain.c 文件中的 libmain() 函数。你首先要修改一下 libmain() 函数，使它能够初始化全局指针 thisenv ，让它指向当前用户环境的 Env 结构体。

然后 libmain() 函数就会调用 umain，这个 umain 程序恰好是 user/hello.c 中被调用的函数。在之前的实验中我们发现，hello.c程序只会打印 "hello, world" 这句话，然后就会报出 page fault 异常，原因就是 thisenv->env\_id 这条语句。现在你已经正确初始化了这个 thisenv的值，再次运行就应该不会报错了。

其实这个练习就是让你通过程序获得当前正在运行的用户环境的 env\_id , 以及这个用户环境所对应的 Env 结构体的指针。 env\_id 我们可以通过调用 sys\_getenvid() 这个函数来获得。那么如何获得它对应的 Env结构体指针呢？

通过阅读 lib/env.h 文件我们知道，env\_id的值包含三部分，第31位被固定为0；第10~30这21位是标识符，标示这个用户环境；第0~9位代表这个用户环境所采用的 Env 结构体，在envs数组中的索引。所以我们只需知道 env\_id 的低 0~9 位，我们就可以获得这个用户环境对应的 Env 结构体了。

1. 练习9

内存保护是操作系统的非常重要的一项功能，它可以防止由于用户程序崩溃对操作系统带来的破坏与影响。操作系统通常依赖于硬件的支持来实现内存保护。操作系统可以让硬件能够始终知晓哪些虚拟地址是有效的，哪些是无效的。当程序尝试去访问一个无效地址，或者尝试去访问一个超出它访问权限的地址时，处理器会在这个指令处终止，并且触发异常，陷入内核态，与此同时把错误的信息报告给内核。如果这个异常是可以被修复的，那么内核会修复这个异常，然后程序继续运行。如果异常无法被修复，则程序永远不会继续运行。

　作为一个可修复异常的例子，让我们考虑一下可自动扩展的堆栈。在许多系统中，内核在初始情况下只会分配一个内核堆栈页，如果程序想要访问这个内核堆栈页之外的堆栈空间的话，就会触发异常，此时内核会自动再分配一些页给这个程序，程序就可以继续运行了。

　系统调用也为内存保护带来了问题。大部分系统调用接口让用户程序传递一个指针参数给内核。这些指针指向的是用户缓冲区。通过这种方式，系统调用在执行时就可以解引用这些指针。但是这里有两个问题：

　1. 在内核中的page fault要比在用户程序中的page fault更严重。如果内核在操作自己的数据结构时出现 page faults，这是一个内核的bug，而且异常处理程序会中断整个内核。但是当内核在解引用由用户程序传递来的指针时，它需要一种方法去记录此时出现的任何page faults都是由用户程序带来的。

　2. 内核通常比用户程序有着更高的内存访问权限。用户程序很有可能要传递一个指针给系统调用，这个指针指向的内存区域是内核可以进行读写的，但是用户程序不能。此时内核必须小心不要去解析这个指针，否则的话内核的重要信息很有可能被泄露。

　　现在你需要通过仔细检查所有由用户传递来指针所指向的空间来解决上述两个问题。当一个程序传递给内核一个指针时，内核会检查这个地址是在整个地址空间的用户地址空间部分，而且页表也运行进行内存的操作。

修改kern/trap.c文件，使其能够实现：当在内核模式下发现页错，trap.c 文件会panic。

提示：

为了能够判断这个page fault是出现在内核模式下还是用户模式下，我们应该检查 tf\_cs 的低几位。

阅读 user\_mem\_assert （在 kern/pmap.c），并且实现 user\_mem\_check;

修改一下 kern/syscall.c 去检查输入参数。

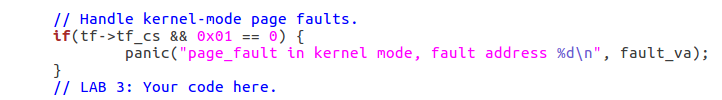
启动内核后，运行 user/buggyhello 程序，用户环境可以被销毁，内核不可以panic，你应该看到：

[00001000] user\_mem\_check assertion failure for va 00000001

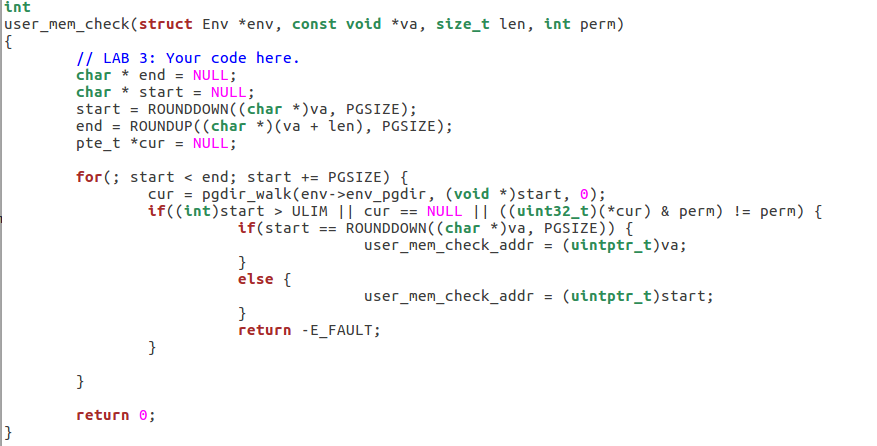
[00001000] free env 00001000

Destroyed the only environment - nothing more to do!

首先我们应该根据什么来判断当前运行的程序时处在内核态下还是用户态下？答案是根据 CS 段寄存器的低2位，这两位的名称叫做 CPL 位，表示当前运行的代码的访问权限级别，0代表是内核态，3代表是用户态。

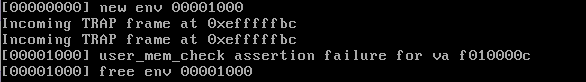
 题目要求我们在检测到这个 page fault 是出现在内核态时，要把这个事件 panic 出来，所以我们把 page\_fault\_handler 文件修改如下：

user\_mem\_check 函数的功能是检查一下当前用户态程序是否有对虚拟地址空间 [va, va+len] 的 perm| PTE\_P 访问权限。自然我们要做的事情应该是，先找到这个虚拟地址范围对应于当前用户态程序的页表中的页表项，然后再去看一下这个页表项中有关访问权限的字段，是否包含 perm | PTE\_P，只要有一个页表项是不包含的，就代表程序对这个范围的虚拟地址没有 perm|PTE\_P 的访问权限。以上就是这段代码的大致思想。

其中syscall中的sys\_cputs函数，这个函数对虚拟地址检查权限。

运行后可以看到

1. 练习10

没有新东西，就是测试练习9的代码

### 实验结果

# 四、xv6实验系统分析

## xv6的系统结构（结合示意图）

## （二）xv6的引导

## （三）xv6的进程与调度分析

## （四）xv6的内存管理

主要的代码文件：

Pmap.c

Pmap.h

Memlayout.h

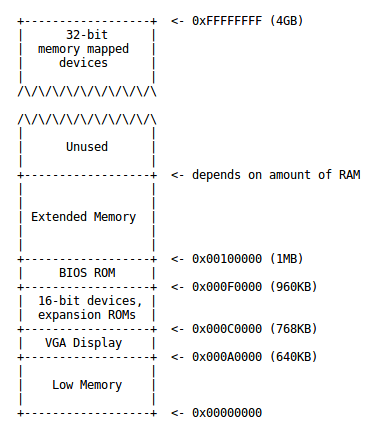
概述：

采用二级页表

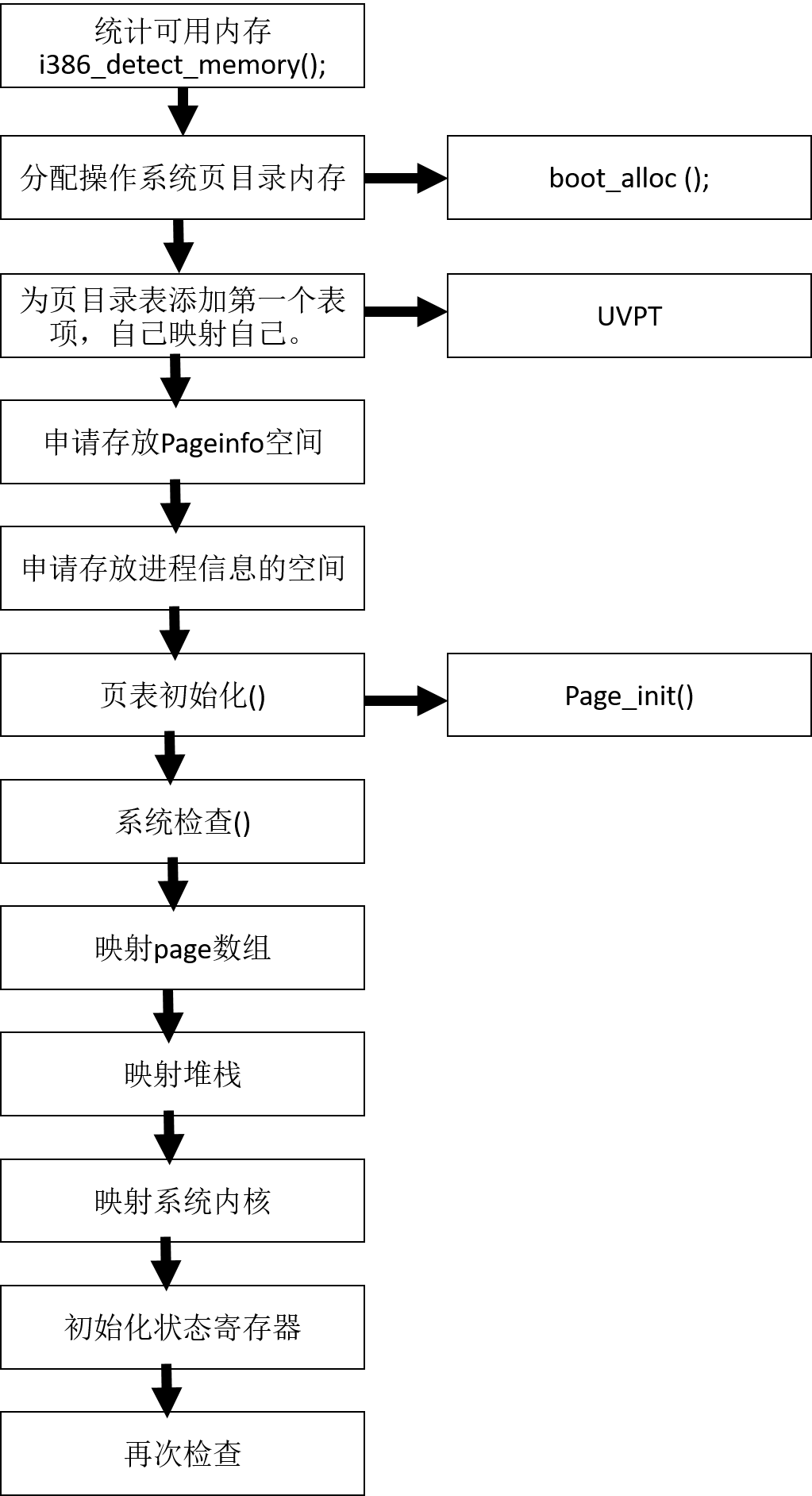
只分页，不分段

最多支持2GB的物理内存

内存格式



一个虚拟地址(Virtual Address)是由两部分组成，一个是段选择子(segment selector)，另一个是段内偏移(segment offset)。一个线性地址(Linear Address)指的是通过段地址转换机构把虚拟地址进行转换之后得到的地址。一个物理地址(Physical Addresses)是分页地址转换机构把线性地址进行转换之后得到的真实的内存地址，这个地址将会最终送到你的内存芯片的地址总线上。

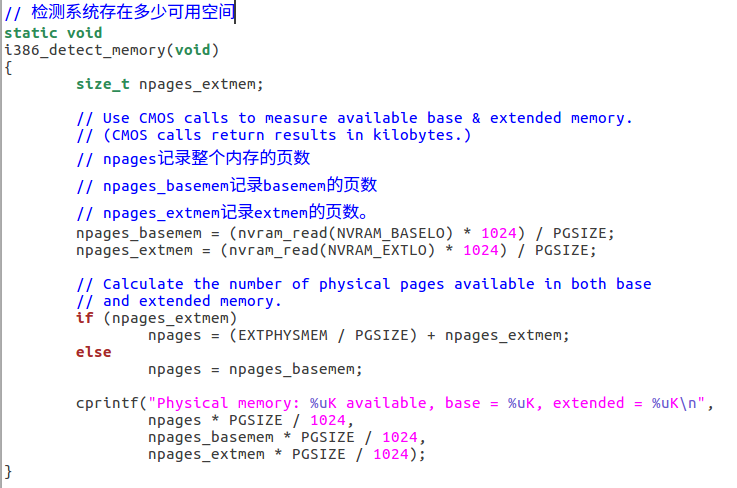


我们首先来看实验中学到的东西。

Jos内核是以页（page）为最小颗粒度来管理内存。它利用了一个结构体Pageinfo的链表来记录哪些页是空闲的，链表中的每一个节点对应一个物理页。

在内核开始的时候，我们首先运行mem\_init()函数，对整个操作系统内存管理系统进行初始化。

他先调用了i386\_detect\_memory，这部分代码是测试有多少可用的内存空间。

之后调用

kern\_pgdir = (pde\_t \*) boot\_alloc(PGSIZE);

memset(kern\_pgdir, 0, PGSIZE);

其中kern\_pgdir是一个指针，pde\_t \*kern\_pgdir，它是指向操作系统的页目录表的指针，操作系统之后工作在虚拟内存模式下时，就需要这个页目录表进行地址转换。我们为这个页目录表分配的内存大小空间为PGSIZE，即一个页的大小。并且首先把这部分内存清0。

Boot\_alloc()这个函数需要实现，这个只是暂时的页分配器。

所以这条kern\_pgdir = (pde\_t \*) boot\_alloc(PGSIZE);指令就会分配一个页的内存，并且这个页就是紧跟着操作系统内核之后。

kern\_pgdir[PDX(UVPT)] = PADDR(kern\_pgdir) | PTE\_U | PTE\_P;

这一条指令就是再为页目录表添加第一个页目录表项。通过查看memlayout.h文件，我们可以看到，UVPT的定义是一段虚拟地址的起始地址，0xef400000，从这个虚拟地址开始，存放的就是这个操作系统的页表kern\_pgdir，所以我们必须把它和页表kern\_pgdir的物理地址映射起来，PADDR(kern\_pgdir)就是在计算kern\_pgdir所对应的真实物理地址。

下一条命令需要我们去补充，这条命令要完成的功能是分配一块内存，用来存放一个struct PageInfo的数组，数组中的每一个PageInfo代表内存当中的一页。操作系统内核就是通过这个数组来追踪所有内存页的使用情况的。

然后初始化page\_init()

要初始化page数组，和pages\_free\_list链表。

检查free\_list。并且把它给反过来。

系统检查page\_alloc() 和 page\_free()能否正常工作。

1. 从free\_page\_list中取出一个空闲页的PageInfo结构体

　 2. 修改free\_page\_list相关信息，比如修改链表表头

　 3. 修改取出的空闲页的PageInfo结构体信息，初始化该页的内存

1. 修改被回收的页的PageInfo结构体的相应信息。

　 2. 把该结构体插入回page\_free\_list空闲页链表。

uintptr\_t -- 表示虚拟地址

physaddr\_t -- 表示物理地址

地址转换

把物理地址转换为虚拟地址，我们可以采用KADDR(pa)指令来获取。其中pa指的是物理地址。

同样的，如果想通过虚拟地址的值求得物理地址的值，我们可以采用PADDR(va)指令。

讲几个函数

pgdir\_walk函数

1. 通过页目录表求得这个虚拟地址所在的页表页对于与页目录中的页目录项地址 dic\_entry\_ptr。

2. 判断这个页目录项对应的页表页是否已经在内存中。

3. 如果在，计算这个页表页的基地址page\_base，然后返回va所对应页表项的地址 &page\_base[page\_off]

4. 如果不在则，且create为true则分配新的页，并且把这个页的信息添加到页目录项dic\_entry\_ptr中。

5. 如果create为false，则返回NULL。

boot\_map\_region函数，函数原型 static void boot\_map\_region(pde\_t \*pgdir, uintptr\_t va, size\_t size, physaddr\_t pa, int perm)，这个函数的功能在注释中被这样解释：　把虚拟地址空间范围[va, va+size)映射到物理空间[pa, pa+size)的映射关系加入到页表pgdir中。这个函数主要的目的是为了设置虚拟地址UTOP之上的地址范围，这一部分的地址映射是静态的，在操作系统的运行过程中不会改变，所以这个页的PageInfo结构体中的pp\_ref域的值不会发生改变。

这个函数要完成的步骤如下：需要完成一个循环，在每一轮中，把一个虚拟页和物理页的映射关系存放到响应的页表项中。直到把size个字节的内存都分配完。

接下来再继续查看page\_insert()，函数原型如下 page\_insert(pde\_t \*pgdir, struct PageInfo \*pp, void \*va, int perm)，功能上是完成：把一个物理内存中页pp与虚拟地址va建立映射关系。这个函数的主要步骤如下：

1. 首先通过pgdir\_walk函数求出虚拟地址va所对应的页表项。

2. 修改pp\_ref的值。

3. 查看这个页表项，确定va是否已经被映射，如果被映射，则删除这个映射。

4. 把va和pp之间的映射关系加入到页表项中。

这里要注意，pp->pp\_ref++这条语句，一定要放在page\_remove之前，这是为了处理一种特殊情况：pp已经映射到va上了。

接下来继续完成page\_lookup()函数，函数原型：struct PageInfo \* page\_lookup(pde\_t \*pgdir, void \*va, pte\_t \*\*pte\_store)， 函数的功能为：

返回虚拟地址va所映射的物理页的PageInfo结构体的指针，如果pte\_store参数不为0，则把这个物理页的页表项地址存放在pte\_store中。

这个函数的功能就很容易实现了，我们只需要调用pgdir\_walk函数获取这个va对应的页表项，然后判断这个页是否已经在内存中，如果在则返回这个页的PageInfo结构体指针。并且把这个页表项的内容存放到pte\_store中。

最后一个就是page\_remove函数，它的原型是：void page\_remove(pde\_t \*pgdir, void \*va)，功能就是把虚拟地址va和物理页的映射关系删除。

注释里面还提示了要注意的几个细节：

1. pp\_ref值要减一

2. 如果pp\_ref减为0，要把这个页回收

3. 这个页对应的页表项应该被置0

## （五）xv6的文件系统

## （六）xv6 的I/O

# 五、总结与心得

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