Lab1 完成文档！

4 月 11 日：
1.1 环境的准备
https://www.cnblogs.com/gatsby123/p/9746193.html

https://blog.csdn.net/qq_42008309/article/details/105196468

至此环境搭建完毕！

1.2 Git 的学习

我的需求：在 ubuntu 下实现代码的管理。
学习了几个命令：
git clone
git status
当添加新的文件或者修改时，使用 git add
提交文件:git commit
最后需要： git push 这时候提示需要填写用户名和密码
https://blog.csdn.net/Chenftli/article/details/81141010

https://blog.csdn.net/qq_38335037/article/details/81163312?utm_source=blogxgwz5
https://blog.csdn.net/u010530712/article/details/79076299?depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromBaidu-1&utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromBaidu-1

# lab1

## Part 1: PC Bootstrap

## Getting Started with x86 assembly

实模式和保护模式
https://blog.csdn.net/rosetta/article/details/8933200?depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromBaidu-1&utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromBaidu-1

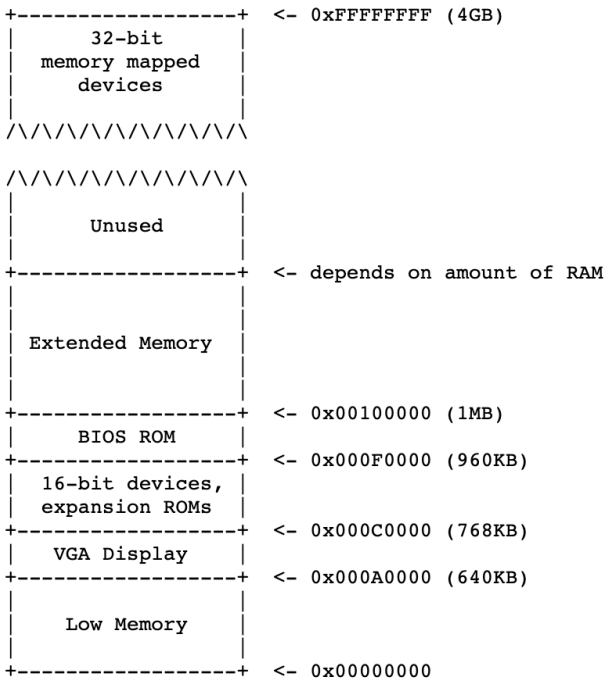
## Exercise1

**Exercise 1.** Familiarize yourself with the assembly language materials available on the 6.828 reference page. You don't have to read them now, but you'll almost certainly want to refer to some of this material when reading and writing x86 assembly.

We do recommend reading the section "The Syntax" in Brennan's Guide to Inline Assembly. It gives a good (and quite brief) description of the AT&T assembly syntax we'll be using with the GNU assembler in JOS.

熟悉汇编语言。

## The PC's Physical Address Space




## The ROM BIOS

## Exercise2

**Exercise 2.** Use GDB's `si` (Step Instruction) command to trace into the ROM BIOS for a few more instructions, and try to guess what it might be doing. You might want to look at Phil Storrs I/O Ports Description, as well as other materials on the 6.828 reference materials page. No need to figure out all the details – just the general idea of what the BIOS is doing first.

熟悉 GDB 调试相关指令

# Part 2: The Boot Loader

我现在的理解　：

首先 BIOS 初始化硬件设备，从启动盘中加载 boot loader， boot loader 包含两个文件： 汇编语言文件 boot.S 和 C 语言文件 main.c 文件， 前者的

作用是将工作模式从实模式转变为保护模式； 后者的作用是加载内核进入内存中。

## Exercise 3

**Exercise 3.** Take a look at the lab tools guide, especially the section on GDB commands. Even if you're familiar with GDB, this includes some esoteric GDB commands that are useful for OS work.

Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in `boot/boot.S`, using the source code and the disassembly file `obj/boot/boot.asm` to keep track of where you are. Also use the `x/i` command in GDB to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the disassembly in `obj/boot/boot.asm` and GDB.

Trace into `bootmain()` in `boot/main.c`, and then into `readsect()`. Identify the exact assembly instructions that correspond to each of the statements in `readsect()`. Trace through the rest of `readsect()` and back out into `bootmain()`, and identify the begin and end of the `for` loop that reads the remaining sectors of the kernel from the disk. Find out what code will run when the loop is finished, set a breakpoint there, and continue to that breakpoint. Then step through the remainder of the boot loader.

#boot.S boot.S 的第一条语句(cli)会被 BIOS 加载到内存地址 0x7c00 处，

BIOS 执行完必要的准备工作后就跳到 0x7c00 执行 boot sector。

boot.S 开始运行时处在 16 位模式下，最后会切换到 32 位保护模式，它的代码非常简洁，也很干脆，流程如下：

使能 A20 地址线
执行 lgdt 加载 gdt
进入保护模式
设置各个段选择子及栈寄存器 esp

调用 main.c 中的 bootmain 函数

#main.c main.c 的入口为 bootmain 函数，bootmain 函数也非常的简洁，流程如下：

读取 kernel 的 ELF 文件头部
依次加载每一个程序段
跳转到 kernel 程序入口去执行

有一些问题：

下面回答一下文中提出的四个问题：

1. 在什么时候处理器开始运行于32bit模式？到底是什么把CPU从16位切换为32位工作模式？

答：在boot.S文件中，计算机首先工作于实模式，此时是16bit工作模式。当运行完 " ljmp $PROT_MODE_CSEG, $protcseg " 语句后，正式进入32位工作模式。根本原因是此时CPU工作在保护模式下。

2. boot loader中执行的最后一条语句是什么？内核被加载到内存后执行的第一条语句又是什么？

答：boot loader执行的最后一条语句是bootmain子程序中的最后一条语句 " ((void (*)(void)) (ELFHDR->e_entry))(); "，即跳转到操作系统内核程序的起始指令处。

这个第一条指令位于/kern/entry.S文件中，第一句 movw $0x1234, 0x472

3. 内核的第一条指令在哪里？

答：上一个问题中已经回答过这个问题，第一条指令位于/kern/entry.S文件中。

4. boot loader是如何知道它要读取多少个扇区才能把整个内核都送入内存的呢？在哪里找到这些信息？

答：首先关于操作系统一共有多少个段，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。这个表中的每个表项分别对应操作系统的一个段。并且每个表项的内容包括这个段的大小，段起始地址偏移等等信息。所以如果我们能够找到这个表，那么就能够通过表项所提供的信息来确定内核占用多少个扇区。

那么关于这个表存放在哪里的信息，则是存放在操作系统内核映像文件的ELF头部信息中。

# Loading the kernel

## Exercise 4

**Exercise 4.** Read about programming with pointers in C. The best reference for the C language is *The C Programming Language* by Brian Kernighan and Dennis Ritchie (known as 'K&R'). We recommend that students purchase this book (here is an Amazon Link) or find one of MIT's 7 copies.

Read 5.1 (Pointers and Addresses) through 5.5 (Character Pointers and Functions) in K&R. Then download the code for pointers.c, run it, and make sure you understand where all of the printed values come from. In particular, make sure you understand where the pointer addresses in printed lines 1 and 6 come from, how all the values in printed lines 2 through 4 get there, and why the values printed in line 5 are seemingly corrupted.

There are other references on pointers in C (e.g., A tutorial by Ted Jensen that cites K&R heavily), though not as strongly recommended.

*Warning:* Unless you are already thoroughly versed in C, do not skip or even skim this reading exercise. If you do not really understand pointers in C, you will suffer untold pain and misery in subsequent labs, and then eventually come to understand them the hard way. Trust us; you don't want to find out what "the hard way" is.

这个练习是熟悉 C 语言指针的题。

以下是给出的练习指针代码示例：

```
#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 = a;
for (i = 0; i < 4; i++)
a[i] = 100 + i;
c[0] = 200;
printf("2: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
a[0], a[1], a[2], a[3]);

c[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]);

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]);

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]);

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;
}
```

```
dingxiaolu@ubuntu:~$ gcc  pointer.c -o a.out
dingxiaolu@ubuntu:~$ ./a.out
1: a = 0x7ffcc09719f0, b = 0xc2a010, c = (nil)
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 = 0x7ffcc09719f0, b = 0x7ffcc09719f4, c = 0x7ffcc09719f1
dingxiaolu@ubuntu:~$ gedit pointer.c
```

You should now be able to understand the minimal ELF loader in `boot/main.c`. It reads each section of the kernel from disk into memory at the section's load address and then jumps to the kernel's entry point.

这句说明了，boot loader 中根据 内核 ELF 文件中的 program Header 中的段地址和大小去加载内核相应的部分进内存中， 并且 jump 到内核的 entry point，将控制权交给内核。

## Exercise 5

**Exercise 5.** Trace through the first few instructions of the boot loader again and identify the first instruction that would "break" or otherwise do the wrong thing if you were to get the boot loader's link address wrong. Then change the link address in `boot/Makefrag` to something wrong, run `make clean`, recompile the lab with `make`, and trace into the boot loader again to see what happens. Don't forget to change the link address back and `make clean` again afterward!

问题重述：

再一次追踪一下 boot loader 的一开始的几句指令，找到第一条满足如下条件的指令处：

当修改了 boot loader 的链接地址，这个指令就会出现错误。

找到这样的指令后，把 boot loader 的链接地址修改一下，我们要在 boot/Makefrag 文件中修改它的链接地址，修改完成后运行 make clean， 然后通过 make 指令重新编译内核，再找到那条指令看看会发生什么。 最后别忘了改回来。

修改 bootloader 的链接地址，链接地址和加载地址的概念：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，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。
在 boot loader 中，由于在 boot loader 运行时还没有任何的分段分页处理机制，所以 boot loader 的链接地址就应该等于加载地址。BIOS 默认的是将 boot loader 加载到 0x7c00 内存地址处，所以就要求 boot loader 的加载地址也在 0x7C00 处。 boot loader 的地址的设定在 boot/Makefrag 中完成，题目要求，需要在其中修改相应部分。
首先按照题目要求，在 lab 目录下输入 make clean，清除掉之前编译出来的内核可执行文件，在清除之前你可以先把 obj/boot/boot.asm 文件拷贝出来，之后可以用来比较。然后打开这个 boot/Makefrag 文件，我们会发现下列语句：

```
$(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
```

其中的-Ttext 0x7C00，就是指定链接地址，我们可以把它修改为 0x7E00，然后保存退出。

然后在 lab 下输入 make，重新编译内核，首先查看一下 obj/boot/boot.asm，并且和之前的那个 obj/boot/boot.asm 文件做比较。下图是新编译出来的 boot.asm：

```
.globl start
start:
  .code16                     # Assemble for 16-bit mode
  cli                         # Disable interrupts
    7e00:   fa                      cli
  cld                         # String operations increment
    7e01:   fc                      cld

  # Set up the important data segment registers (DS, ES, SS).
  xorw    %ax,%ax             # Segment number zero
    7e02:   31 c0                   xor     %eax,%eax
```

下图是修改之前的boot.asm

```
.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
```

可以看出，二者区别在于可执行文件中的链接地址不同了，原来是从 0x7C00 开始，现在则是从 0x7E00 开始。

然后我们还是按照原来的方式，调试一下内核：

由于 BIOS 会把 boot loader 程序默认装入到 0x7c00 处，所以我们还是再 0x7C00 处设置断点，并且运行到那里，结果发现如下：

```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Breakpoint 1, 0x00007c00 in ?? ()
(gdb)
```

可见第一条执行的指令仍旧是正确的，所以我们接着往下一步步运行。

接下来的几步仍旧是正常的，但是直到运行到一条指令：

```
[   0:7c1e] => 0x7c1e:  lgdtw  0x7e64
0x00007c1e in ?? ()
(gdb) x/6xb 0x7e64
0x7e64: 0x00    0x00    0x00    0x00    0x00    0x00
(gdb) x/6xb 0x7c64
0x7c64: 0x17    0x00    0x4c    0x7e    0x00    0x00
```

图中的 0x7c1e 处指令，

lgdtw 0x7e64

这条指令我们之前讲述过，是把指令后面的值所指定内存地址处后 6 个字节的值输入全局描述符表寄存器 GDTR，但是当前这条指令读取的内存地址是 0x7e64，我们在图中也展示了一下这个地址处后面 6 个单元存放的值，发现是全部是 0。这肯定是不对的，正确的应该是在 0x7c64 处存放的值，即图中最下面一样的值。**可见，问题出在这里，GDTR 表的值读取不正确，这是实现从实模式到保护模式转换的非常重要的一步。**

我们可以继续运行，知道发现下面这句：

```
[   0:7c2d] => 0x7c2d:  ljmp   $0x8,$0x7e32
0x00007c2d in ?? ()
(gdb)
[f000:e05b]    0xfe05b: cmpl   $0x0,%cs:0x6ac8
```

正常来说，0x7c2d 处的指令

ljmp $0x08m $0x7e32

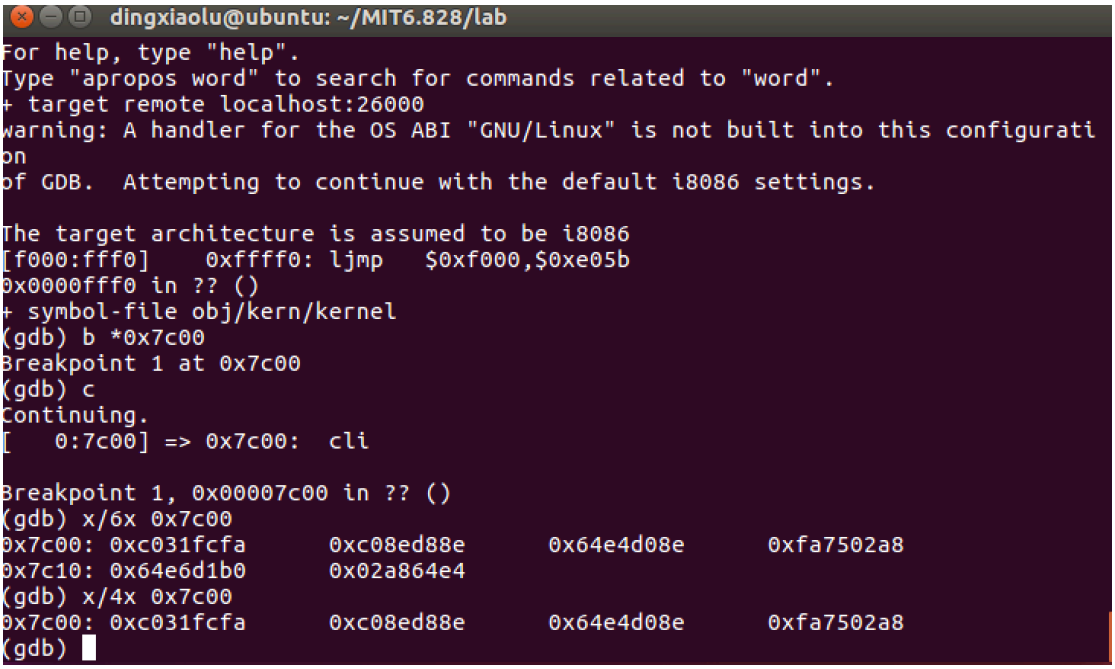
应该跳转到的地址应该就是 ljmp 的下一条指令地址，即 0x7c32，但是这里给的值是 0x7e32，所以造成错误，此时下条指令变成了 0xfe05b。

## Exercise 6

**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/Nx 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.)

这个题目我感觉就是学习 x/Nx ADDR 的 GDB 的调试语句，我通过在 boot loader 的链接地址处打了一个断点，然后在断点 0x7c00 处看其后 4 个字 里面存储的数据。




# Part 3: The Kernel

# Using virtual memory to work around position dependence

## Exercise 7

**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 the `movl %eax, %cr0` in `kern/entry.S`, trace into it, and see if you were right.

内核文件的入口地址是 0x10000c 处

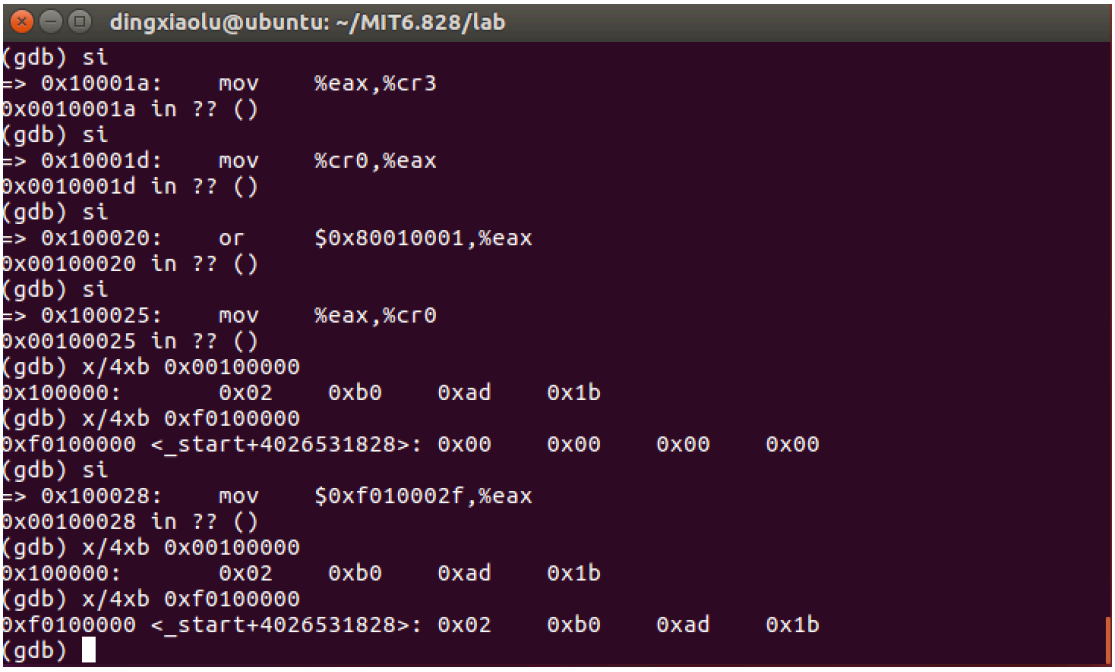
```
dingxiaolu@ubuntu:~/MIT6.828/lab$ readelf -l obj/kern/kernel

Elf file type is EXEC (Executable file)
Entry point 0x10000c
There are 3 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0xf0100000 0x00100000 0x07120 0x07120 R E 0x1000
  LOAD           0x009000 0xf0108000 0x00108000 0x0a948 0x0a948 RW  0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10

 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata .stab .stabstr
   01     .data .bss
   02
```

到达断点后，单步执行，直到运行到 mov %eax，%cr0 指令时，我们发现地址 0x100000 和地址 0xf0100000 内存中存放的数据是不同的，然后继续输入 si 命令，可以发现这两个内存地址中的存放的数据是相同的。




可见原本存放在 0xf0100000 处的内容，已经被映射到 0x00100000 处了。

在第二小问中，将 movl %eax,%cr0 这一句给注释掉，然后重新编译。

```
.globl entry
entry:
        movw    $0x1234,0x472                   # warm boot

        # We haven't set up virtual memory yet, so we're running from
        # the physical address the boot loader loaded the kernel at: 1MB
        # (plus a few bytes).  However, the C code is linked to run at
        # KERNBASE+1MB.  Hence, we set up a trivial page directory that
        # translates virtual addresses [KERNBASE, KERNBASE+4MB) to
        # physical addresses [0, 4MB).  This 4MB region will be
        # sufficient until we set up our real page table in mem_init
        # in lab 2.

        # Load the physical address of entry_pgdir into cr3.  entry_pgdir
        # is defined in entrypgdir.c.
        movl    $(RELOC(entry_pgdir)), %eax
        movl    %eax, %cr3
        # Turn on paging.
        movl    %cr0, %eax
        orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
        #movl   %eax, %cr0

        # Now paging is enabled, but we're still running at a low EIP
        # (why is this okay?).  Jump up above KERNBASE before entering
        # C code.
        mov     $relocated, %eax
        jmp     *%eax
```

继续仿真：

```
dingxiaolu@ubuntu: ~/MIT6.828/lab
=> 0x100015:    mov    $0x110000,%eax
0x00100015 in ?? ()
(gdb) si
=> 0x10001a:    mov    %eax,%cr3
0x0010001a in ?? ()
(gdb) si
=> 0x10001d:    mov    %cr0,%eax
0x0010001d in ?? ()
(gdb) si
=> 0x100020:    or     $0x80010001,%eax
0x00100020 in ?? ()
(gdb) si
=> 0x100025:    mov    $0xf010002c,%eax
0x00100025 in ?? ()
(gdb) si
=> 0x10002a:    jmp    *%eax
0x0010002a in ?? ()
(gdb) si
=> 0xf010002c <relocated>:      add    %al,(%eax)
relocated () at kern/entry.S:74
74              movl    $0x0,%ebp                       # nuke frame pointer
(gdb) si
Remote connection closed
(gdb)
```

在 qemu 中报错：

内存越界

```
dingxiaolu@ubuntu:~/MIT6.828/lab$ make qemu-gdb
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tmpl > .gdbinit
***
*** Now run 'make gdb'.
***
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -
serial mon:stdio -gdb tcp::26000 -D qemu.log  -S
VNC server running on `127.0.0.1:5900'
qemu: fatal: Trying to execute code outside RAM or ROM at 0xf010002c

EAX=f010002c EBX=00010094 ECX=00000000 EDX=0000009d
ESI=00010094 EDI=00000000 EBP=00007bf8 ESP=00007bec
EIP=f010002c EFL=00000086 [--S--P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
CS =0008 00000000 ffffffff 00cf9a00 DPL=0 CS32 [-R-]
SS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
DS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
FS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
GS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
```

分析：

在地址 0x100025 处，需要将 eax 的寄存器设置为 0xf010002c 的位置。

```
=> 0x100025:    mov    $0xf010002c,%eax
0x00100025 in ?? ()
```

当执行到：jmp 指令时：

```
=> 0x10002a:    jmp    *%eax
0x0010002a in ?? ()
(gdb) si
=> 0xf010002c <relocated>:      add    %al,(%eax)
relocated () at kern/entry.S:74
74              movl    $0x0,%ebp                       # nuke frame pointer
(gdb) si
Remote connection closed
(gdb)
```

其中在 0x10002a 处的 jmp 指令，要跳转的位置是 0xf010002C，由于没有进

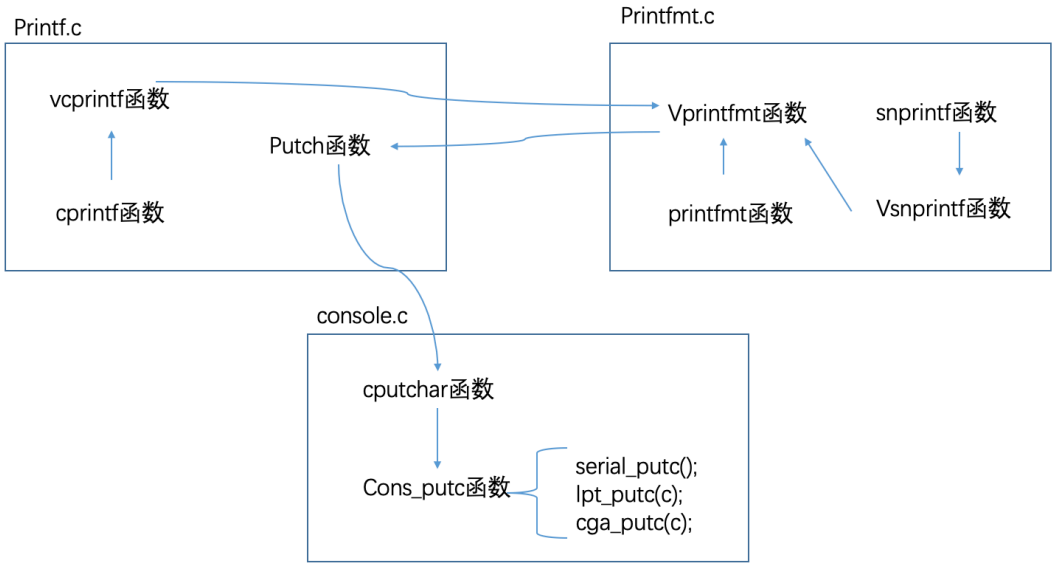
行分页管理，此时不会进行虚拟地址到物理地址的转化。所以报出错误。

## 结论

movl %eax ，%cr0
这条指令的作用就是开启分页模式，在执行这条语句之前，所有的线性地址直接等于物理地址，执行之后，线性地址需要经过 MMU 的映射才对应到物理地址。
所以执行之前 0x100000 处为内核代码，0xf0100000 处为空。执行之后 0x100000 和 0xf0100000 都映射到物理内存 0x100000，所以他们的内容相同。

## Formatted Printing to the Console

阅读 kern/printf.c, lib/printfmt.c, and kern/console.c 代码；理清关系如下。




分析：
这些函数最终都会调用到 cputchar()，cputchar()打印一个字符到屏幕。cputchar()会调到 kern/console.c 中的 cga_putc()，该函数将 int c 打印到控制台，可以看到该函数处理会打印正常的字符外，还能处理回车换行等控制字符，甚至还能处理滚屏。cga_putc()会将字符对应的 ascii 码存储到 crt_buf[crt_pos]处，实际上 crt_buf 在初始化的时候被初始为

```
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;
}
```

所以往控制台写字符串，本质还是往物理地址 0xB8000 开始的显存写数据。

根据函数调用图，可以发现真正实现字符串输出的是 vprintfmt()函数，其他函数都是对它的包装。

由 printfmt.c 文件中的注释可知，这个文件中定义的子程序是我们能在编程时直接利用 printf 函数向屏幕输出信息的关键。

```
// 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.
```

## Exercise 8

**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.

写出自己的 code，让输出可以处理 8 进制！

在 vprintfmt 函数中格式化的处理。

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

// (unsigned) octal
case 'o':
        // Replace this with your code.
        putch('X', putdat);
        putch('X', putdat);
        putch('X', putdat);
        break;
```

找到 case ‘o’ 处，修改其代码为：

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

// (unsigned) octal
case 'o':
        // Replace this with your code.
        //putch('X', putdat);
        //putch('X', putdat);
        //putch('X', putdat);
        //break;
        num = getuint(&ap, lflag);
        if((long long) num<0){
                putch('-',putdat);
                num = -(long long) num;
        }
        base = 8;
        goto number;
```

保存后，重新 make ，运行 ./grade-lab1.

```
dingxiaolu@ubuntu:~/MIT6.828/lab$ ./grade-lab1
running JOS: (1.1s)
  printf: OK
  backtrace count: FAIL
    AssertionError: got:
      0
    expected:
      8

  backtrace arguments: FAIL
    AssertionError: got:

    expected:
      00000000
      00000000
      00000001
      00000002
      00000003
      00000004
      00000005

  backtrace symbols: FAIL
    AssertionError: got:

    expected:
      test_backtrace
      test_backtrace
      test_backtrace
      test_backtrace
      test_backtrace
      test_backtrace
      i386_init

  backtrace lines: FAIL
```

发现 printf:ok，说明成功！

回答后面几个问题：

1）

Explain the interface between printf.c and console.c. Specifically, what function does console.c export? How is this function used by printf.c?

答：根据关系图可以看出它们之间的关系。

2）解释代码的含义

Explain the following from console.c:

////判断是否需要滚屏。文本模式下一页屏幕最多显示 25*80 个字符，

```
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       }
```

答：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。

这段代码的意思是当屏幕输出满了以后，将屏幕上的内容都向上移一行，即将第一行移出屏幕，同时将最后一行用空格填充，最后将光标移动到屏幕最后一行的开始处。

3）

For the following questions you might wish to consult the notes for Lecture 2. These notes cover GCC's calling convention on the x86.

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?
- 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.

在 monitor.c 文件处添加这一句。

```
void
monitor(struct Trapframe *tf)
{
        char *buf;

        cprintf("Welcome to the JOS kernel monitor!\n");
        cprintf("Type 'help' for a list of commands.\n");


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

执行 make qemu：

```
dingxiaolu@ubuntu:~/MIT6.828/lab$ make qemu
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tmpl > .gdbinit
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -serial mon
:stdio -gdb tcp::26000 -D qemu.log
VNC server running on `127.0.0.1:5900'
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
kernel panic at kern/pmap.c:128: mem_init: This function is not finished

Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
x 1, y 3, z 4
K>
```

4）运行，这是个大小端的表示

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.

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?

57616 的 16 进制为 e110 ，所以前面就变成 He110 ； 0x00646c72 按照%s 打印的话，X86 是小端存储， 则 0x00646c72，存放在 0x00，0x01，0x02，0x03 处分别为 0x72，0x6c，0x64，0x00. 分别为 ‘r’ ， ‘l’ ， ‘d’ ， ‘\0’ ，则后面就是 World。

整体输出就是 He110，World

```
dingxiaolu@ubuntu:~/MIT6.828/lab$ make qemu
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -
serial mon:stdio -gdb tcp::26000 -D qemu.log
VNC server running on `127.0.0.1:5900'
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
kernel panic at kern/pmap.c:128: mem_init: This function is not finished

Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
He110 WorldK>
```

最后可以发现输出：

```
Type 'help'
He110 World
```

。

5）

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);
```

```
void
monitor(struct Trapframe *tf)
{
        char *buf;

        cprintf("Welcome to the JOS kernel monitor!\n");
        cprintf("Type 'help' for a list of commands.\n");

        //test
        unsigned int i = 0x00646c72;
        cprintf("H%x Wo%s \n", 57616, &i);

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

输出

```
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
He110 World
x=3,y=-267321476
```

X 是指定值，y 则是一个随机值。

# The Stack

## Exercise 9

**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?

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？以及这个堆栈处于内存的什么位置？ 内核是如何为它的栈保留空间的？ 堆栈指针又是指向这块被保留的区域的哪一端？

解题：

**1）考虑操作系统内核是从哪条指令开始去初始化它的堆栈空间的？**

首先，PC 启动时，BIOS 启动会去初始化系统硬件信息。最后调用磁盘启动盘，执行 Boot Loader 的两部分代码，boot.S 和 main.c 文件，boot.S 文件的作用是将系统从实模式转变为保护模式，main.c 文件的作用是加载内核到物理地址中。当 main.c 文件中 bootmain 函数运行到最后时，它执行的最后一条语句是：

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

跳转到 entry.S 文件的 entry 地址处。此时控制权完全交给内核了。
在跳转到 entry.S 之前并没有对%esp 和%ebp 寄存器进行修改，可见在 bootmain 函数中并没有对其进行初始化。
在 entry.S 文件的最后一句是：
call i386_init　　调用 i386_init()函数。这个函数存在于 init.c 文件中。在这个程序中已经开始对操作系统进行一系列的初始化工作了，并且进入 mointor 函数中。可见到 i386_init 子程序时，堆栈已经设置好了。

```
# Clear the frame pointer register (EBP)
# so that once we get into debugging C code,
# stack backtraces will be terminated properly.
movl    $0x0,%ebp                       # nuke frame pointer

# Set the stack pointer
movl    $(bootstacktop),%esp

# now to C code
call    i386_init
```

从图中可以看到，在调用 i386_init 之前的两条语句已经对 esp 和 ebp 寄存器进行了初始化。

**2）以及这个堆栈处于内存的什么位置？**

通过查看反汇编文件 boot.asm，查看进入 entry.S 的入口地址：

```
    // call the entry point from the ELF header
    // note: does not return!
    ((void (*)(void)) (ELFHDR->e_entry))();
7d6b:       ff 15 18 00 01 00       call   *0x10018
```

设置断点在 0x7d6b。接着单步调试，发现进入 entry.S 的第一个指令是 movw $0x1234,0x472　　指令地址为 0x10000c，这个比较好理解，因为在 bootmain 里面，我们已经把操作系统的内核文件全部加载到物理内存 0x100000 处了。
看 kernel.asm 反汇编文件：

```
Disassembly of section .text:

f0100000 <_start+0xeffffff4>:
.globl          _start
_start = RELOC(entry)

.globl entry
entry:
        movw    $0x1234,0x472                   # warm boot
f0100000:       02 b0 ad 1b 00 00       add    0x1bad(%eax),%dh
f0100006:       00 00                   add    %al,(%eax)
f0100008:       fe 4f 52                decb   0x52(%edi)
f010000b:       e4                      .byte 0xe4

f010000c <entry>:
f010000c:       66 c7 05 72 04 00 00    movw   $0x1234,0x472
f0100013:       34 12
        # sufficient until we set up our real page table in mem_init
        # in lab 2.
```

至于这边为什么 entry 的起始地址是 f010000c，因为这是虚拟地址。具体看下面分析。

所以 0x10000C 是系统内核的第一条指令所在的物理地址处。

```
dingxiaolu@ubuntu: ~/MIT6.828/lab
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
+ target remote localhost:26000
warning: A handler for the OS ABI "GNU/Linux" is not built into this configurati
on
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) b *0x7d6b
Breakpoint 1 at 0x7d6b
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d6b:      call   *0x10018

Breakpoint 1, 0x00007d6b in ?? ()
(gdb) si
=> 0x10000c:    movw   $0x1234,0x472
0x0010000c in ?? ()
(gdb)
```

继续单步调试：

```
dingxiaolu@ubuntu: ~/MIT6.828/lab
0x00100015 in ?? ()
(gdb) si
=> 0x10001a:    mov    %eax,%cr3
0x0010001a in ?? ()
(gdb) si
=> 0x10001d:    mov    %cr0,%eax
0x0010001d in ?? ()
(gdb) si
=> 0x100020:    or     $0x80010001,%eax
0x00100020 in ?? ()
(gdb) si
=> 0x100025:    mov    %eax,%cr0
0x00100025 in ?? ()
(gdb) si
=> 0x100028:    mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) si
=> 0x10002d:    jmp    *%eax
0x0010002d in ?? ()
(gdb) si
=> 0xf010002f <relocated>:      mov    $0x0,%ebp
relocated () at kern/entry.S:74
74              movl    $0x0,%ebp                       # nuke frame pointer
(gdb)
```

当执行完 jmp *%eax 后，指令地址变为

```
=> 0xf010002f <relocated>:
```

图中的地址是 0xf010002f，很明显这是一个虚拟地址，它的真实地址应该是 0x0010002f，因为所有的内核代码都实际存放在这个内存区域中。之所以现在要把指令地址设置为 0xf010002f，即把操作系统的代码的虚拟地址设置为从 0xf0100000 开始。目的就是能够让程序员在编程时，能够利用虚拟地址空间的低地址空间。如果它编写的程序调用了操作系统的代码，则操作系统代码的虚拟地址一定位于高地址空间 0xf0100000 处。

所以必须有一种机制能够实现，即便程序员在程序中指定的操作系统的代码的虚拟地址在 0xf0100000 高地址空间，但是我们这个机制也能够把这个高地址转换为这个代码真实的在内存中的位置。比如上图中，我们想访问 0xf010002f 处的指令，这是个虚拟地址，当实际运行时，会有一种机制把这个地址转换为真实地址 0x0010002f。

在 entry.S 文件中：

```
# Load the physical address of entry_pgdir into cr3.  entry_pgdir
# is defined in entrypgdir.c.
movl    $(RELOC(entry_pgdir)), %eax
movl    %eax, %cr3
# Turn on paging.
movl    %cr0, %eax
orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
movl    %eax, %cr0
```

第一个语句是写一个 页表，entry_pgdir。这个手写的页表可以自动的把[0xf0000000-0xf0400000]这 4MB 的虚拟地址空间映射为[0x00000000-

0x00400000]的物理地址空间。可见这个页表的映射能力还是比较有限的，只能映射一个区域。对于当前执行的这些指令，这个映射空间就已经足够了。因为当前运行的是内核程序，他们的虚拟空间地址范围在[0xf0000000-0xf0400000]之内。但是当操作系统真正正常的运行起来的时候，这个映射就不够用了。必须采用更全面的，也就是在 lab 2 中要介绍的页表机制。所以当操作系统真正正常运行起来时，entry_pgdir 这个页表将不会再使用。

现在依次分析上述语句，首先看第 1 句，它的功能是把 entry_pgdir 这个页表的起始物理地址送给%eax，这里 RELOC 宏的功能是计算输入参数的物理地址。

第 2 句，把 entry_pgdir 这个页表的起始地址传送给寄存器%cr3。

控制寄存器 cr2 和 cr3 都是和分页机制相关的寄存器。其中 cr3 寄存器存放页表的物理起始地址。

第 3~5 句，修改 cr0 寄存器的值，把 cr0 的 PE 位，PG 位, WP 位都置位 1。其中 PE 位是启用保护标识位，如果被置 1 代表将会运行在保护模式下。PG 位是分页标识位，如果这一位被置 1，则代表开启了分页机制。WP 位是写保护标识，如果被置位为 1，则处理器会禁止超级用户程序向用户级只读页面执行写操作。

这条指令过后，就开始工作在具有分页机制的模式之下了。接下来的指令就可以指定[0xf0000000-0xf0400000]范围的指令了。

然后下面两条指令就把当前运行程序的地址空间提高到[0xf0000000-0xf0400000]范围内。

```
=> 0x100028:    mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) si
=> 0x10002d:    jmp    *%eax
0x0010002d in ?? ()
(gdb) si
=> 0xf010002f <relocated>:      mov    $0x0,%ebp
relocated () at kern/entry.S:74
74              movl    $0x0,%ebp                       # nuke frame pointer
(gdb)
```

可见 relocated 的值为 0xf010002f。此时分页系统会把这个虚拟地址，转换为真实的物理地址。

接着下面两句就是设置堆栈的　ebp 寄存器和 esp 寄存器。

```
# Clear the frame pointer register (EBP)
# so that once we get into debugging C code,
# stack backtraces will be terminated properly.
movl    $0x0,%ebp                       # nuke frame pointer

# Set the stack pointer
movl    $(bootstacktop),%esp

# now to C code
call    i386_init
```

这两个指令分别设置了%ebp，%esp 两个寄存器的值。其中%ebp 被修改为 0。%esp 则被修改为 bootstacktop 的值。这个值为 0xf0110000。另外在 entry.S 的末尾还定义了一个值，bootstack。注意，在数据段中定义栈顶 bootstacktop 之前，首先分配了 KSTKSIZE 这么多的存储空间，专门用于堆栈，这个 KSTKSIZE = 8 * PGSIZE  = 8 * 4096 = 32KB。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。 一共 32kb。

在 entry.S 末尾可以看到这边对 bootstack 的定义。

```
.data
###################################################################
# boot stack
###################################################################
        .p2align        PGSHIFT         # force page alignment
        .globl          bootstack
bootstack:
        .space          KSTKSIZE
        .globl          bootstacktop
bootstacktop:
```

**3，内核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，堆栈指针又是指向这块被保留的区域的哪一端？**

堆栈由于是向下生长的，所以堆栈指针自然要指向最高地址了。最高地址就是我们之前看到的 bootstacktop 的值。所以将会把这个值赋给堆栈指针寄存器。

# Exercise 10

**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.

分析：

首先看 test_backtrace 子程序的入口地址如下：

```
// Test the stack backtrace function (lab 1 only)
void
test_backtrace(int x)
{
f0100040:       55                      push   %ebp
f0100041:       89 e5                   mov    %esp,%ebp
f0100043:       53                      push   %ebx
f0100044:       83 ec 0c                sub    $0xc,%esp
f0100047:       8b 5d 08                mov    0x8(%ebp),%ebx
        cprintf("entering test_backtrace %d\n", x);
f010004a:       53                      push   %ebx
f010004b:       68 80 18 10 f0          push   $0xf0101880
f0100050:       e8 a3 08 00 00          call   f01008f8 <cprintf>
        if (x > 0)
f0100055:       83 c4 10                add    $0x10,%esp
f0100058:       85 db                   test   %ebx,%ebx
f010005a:       7e 11                   jle    f010006d <test_backtrace+0x2d>
                test_backtrace(x-1);
f010005c:       83 ec 0c                sub    $0xc,%esp
f010005f:       8d 43 ff                lea    -0x1(%ebx),%eax
f0100062:       50                      push   %eax
f0100063:       e8 d8 ff ff ff          call   f0100040 <test_backtrace>
f0100068:       83 c4 10                add    $0x10,%esp
f010006b:       eb 11                   jmp    f010007e <test_backtrace+0x3e>
        else
                mon_backtrace(0, 0, 0);
f010006d:       83 ec 04                sub    $0x4,%esp
f0100070:       6a 00                   push   $0x0
f0100072:       6a 00                   push   $0x0
f0100074:       6a 00                   push   $0x0
f0100076:       e8 f3 06 00 00          call   f010076e <mon_backtrace>
f010007b:       83 c4 10                add    $0x10,%esp
        cprintf("leaving test_backtrace %d\n", x);
f010007e:       83 ec 08                sub    $0x8,%esp
f0100081:       53                      push   %ebx
f0100082:       68 9c 18 10 f0          push   $0xf010189c
f0100087:       e8 6c 08 00 00          call   f01008f8 <cprintf>
}
f010008c:       83 c4 10                add    $0x10,%esp
f010008f:       8b 5d fc                mov    -0x4(%ebp),%ebx
f0100092:       c9                      leave
f0100093:       c3                      ret
```

在每次进入 test_backtrace()函数时，

```
f0100040:       55                      push   %ebp
f0100041:       89 e5                   mov    %esp,%ebp
f0100043:       53                      push   %ebx
f0100044:       83 ec 0c                sub    $0xc,%esp
```

这 4 个指令表示，当前函数将其调用函数的栈帧保存起来，并且为当前函数设置新的栈帧。栈帧就是这个程序的开始位置。在前面的分析中，当进入 i386_init 函数，调用 test_backtrace 函数，首先需要先压入 i386_init 的栈帧进入栈，然后设置当前 test_backtrace 的栈帧。

大体的感觉如下：

1）执行 call 指令前，函数调用者将参数入栈，按照函数列表从右到左的顺序

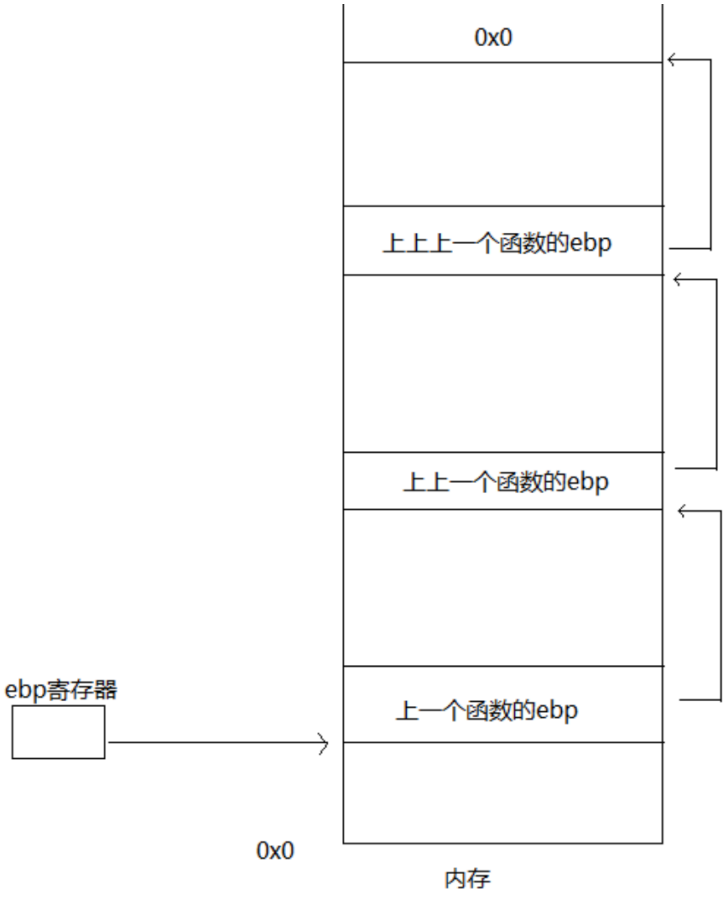
入栈

2）call 指令会自动将当前 eip 入栈，ret 指令将自动从栈中弹出该值到 eip 寄存器

3）被调用函数负责：将 ebp 入栈，esp 的值赋给 ebp。所以反汇编一个函数会发现开头两个指令都是：

push %ebp

mov %esp,%ebp




栈帧的计算方法：

一个栈帧(stack frame)的大小计算如下：

1. 在执行 call test_backtrace 时有一个副作用就是压入这条指令下一条指令的地址，压入 4 字节返回地址

2. push %ebp，将上一个栈帧的地址压入，增加 4 字节

3. push %ebx，保存 ebx 寄存器的值，增加 4 字节

4. sub $0xc（14）,%esp，开辟 20 字节的栈空间，后面的函数调用传参直接操作这个栈空间中的数，而不是用 push 的方式压入栈中

## Exercise 11

**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.

理解：int *p=（int *）100 ：

int *p=(int *)100;这样说吧，这条语句没有任何错误，只是平时谁也不这么用而已。这就是把100这个十进制整型数强制变成了地址值00000000 00000000 00000000 01100100赋给了整型指针(能指向整型数的指针)p。printf("%d",p)和printf("%d",*p)语法上都对，前者是输出指针变量p自身的值，它就是上面提到的十进制数100，若用printf("%X",p)输出的话就是上面那串二进制数的十六进制00000064。后者*p是取p指向的目标内容，也就是地址100中的内容；现在的问题是你没有给100这个地址中放内容，而且100这个地址是不是系统允许用户操作也未知；如果允许操作，那么在int *p=(int *)100; 后加一句*p=123(随便一个数)，就会打出123这个数来；如果不允许操作，那仍然要报错……

在 inc/x86.h 文件中提供了 read_ebp()函数，可以让我们方便获取寄存器 ebp 的值。

```
210 static inline uint32_t
211 read_ebp(void)
212 {
213     uint32_t ebp;
214     asm volatile("movl %%ebp,%0" : "=r" (ebp));
215     return ebp;
216 }
```

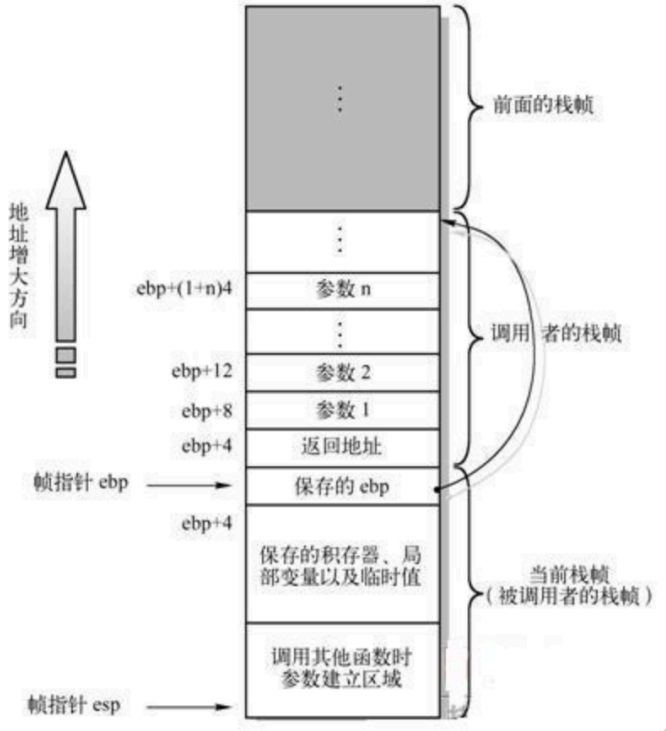
```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
        // Your code here.

        //return 0;
        uint32_t ebp,*p;
        ebp=read_ebp();     //获取当前ebp的值
        while(ebp!=0){
                p=(uint32_t *)ebp;  //将ebp类型转换为指针，赋值给p。
                cprintf("ebp %x eip %x args %08x %08x %08x %08x %08x \n",ebp,p[1],p[2],p[3],p[4],p
[5],p[6]);
                ebp=p[0];
        }
        return 0;
}
```

思路：

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。




从图中可以看出，当前的内存中包含两个栈帧，一个是当前栈帧，即被调用者的栈帧；另一个是调用者的栈帧。其中我们这个函数的功能就是要得到当前栈帧 ebp 寄存器的值，以及调用者栈帧中的返回地址，传递给当前栈帧的输入参数。

## Exercise 12

到目前为止，编写的 backtrace 函数应该能够把导致 mon_backtrace()函数执行的所有函数的地址信息打印出来了。但是在实际情况中，你经常会向弄清楚这些地址对应的到底是哪个函数。

**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_*`
- run `objdump -h obj/kern/kernel`
- run `objdump -G obj/kern/kernel`
- 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.
- 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).

按照题目要求，在 kdebug.c 文件中，修改如下

```
// 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;
}
```

在 monitor.c 文件中，丰富相关代码：

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

        //return 0;
        uint32_t ebp,*p;
        uint32_t eip;
        struct Eipdebuginfo info;
        ebp=read_ebp();    //获取当前ebp的值
        while(ebp!=0){
                p=(uint32_t *)ebp;  //将ebp类型转换为指针，赋值给p。
                //cprintf("ebp %x eip %x args %08x %08x %08x %08x %08x \n",ebp,p[1],p[2],p[3],p
[4],p[5],p[6]);
                eip=p[1];
                cprintf("ebp %x eip %x args %08x %08x %08x %08x %08x \n",ebp,eip,p[2],p[3],p[4],p
[5],p[6]);
                //print file infomation
                debuginfo_eip(eip,&info);
                int fn_offset=eip-info.eip_fn_addr;
                cprintf("%s:%d: %.*s+%d
\n",info.eip_file,info.eip_line,info.eip_fn_namelen,info.eip_fn_name,eip-info.eip_fn_addr);

                ebp=p[0];
        }
        return 0;
}
```

得到如下结果：

```
K> backtrace
ebp f010ff28 eip f0100895 args 00000001 f010ff40 00000000 f010ffcc f0112540
kern/monitor.c:142: monitor+256
ebp f010ff98 eip f01000dd args 00000000 f010ffcc 00000080 f01008e6 00000280
kern/init.c:69: _panic+82
ebp f010ffb8 eip f0100965 args f0101fac 00000080 f0101f80 0001fd80 00010094
kern/pmap.c:236: page_init+0
ebp f010ffd8 eip f0100079 args f0101a20 00001aac 00000640 00000000 00000000
kern/init.c:30: i386_init+57
ebp f010fff8 eip f010003e args 00111021 00000000 00000000 00000000 00000000
kern/entry.S:83: <unknown>+0
K>
```

## 最后测试 lab1，表示全部通过！

```
dingxiaolu@ubuntu:~/6.828/lab$ ./grade-lab1
running JOS: (1.0s)
  printf: OK
  backtrace count: OK
  backtrace arguments: OK
  backtrace symbols: OK
  backtrace lines: OK
Score: 50/50
```

参考：https://blog.csdn.net/bysui/category_6232831.html

https://github.com/clpsz/mit-jos-2014/tree/master/Lab1
https://www.cnblogs.com/gatsby123/p/9759153.html