Lab1

**练习2：**

设置ss和esp寄存器的值，进入保护模式（设置cr0寄存器）

**练习3：**

Boot.s：初始化寄存器，从实模式跳到虚模式，最后调用bootmain函数

Readsect函数：等待磁盘，输出扇区数目和地址信息到端口，读取扇区数据。

循环：

// C Code: // for (; ph < eph; ph++) // readseg(ph->p\_pa, ph->p\_memsz, ph->p\_offset);

7d51: 39 f3 cmp %esi,%ebx

7d53: 73 16 jae 7d6b <bootmain+0x56>

结束语句：将三个参数压栈，调用readseg，esp自增12（清除三个参数），回到循环开始。

结束后执行：7d6b: ff 15 18 00 01 00 call \*0x10018，调用ELF文件的入口函数。

问题：

1. boot.s中的.code32开始执行。Ljmp指令使得处理区切换到保护模式。通过设置cr0寄存器的PE位（是否开启保护模式）和PG位（使用分段还是分页式）。
2. 7d6b: ff 15 18 00 01 00 call \*0x10018

0x10000c: movw $0x1234,0x472

1. 0x10000c
2. ELF文件头中包含响应的信息。

**练习4：**

**练习5：**

Lgdt gdtdesc 和 ljmp $PROT\_MODE\_CSEG, $protcseg会出错，这两句需要计算地址。

**练习6：**

刚进入bootloader时都为0.

在for循环时被修改。

**练习7：**

执行前后者全为0，执行后和前者相同，地址映射完成。

**练习8：**

回答：

1. Printf.c中的putch函数调用了console.c中的cputchar函数。

cprintf->vcprintf->putch->cputchar

1. 当屏幕写满时将内容上移一行，用黑色空格填满最后一行。

CRT\_COLS是显示器每行的字长（1个字占2字节），取值为80；CRT\_ROWS是显示器的行数，取值为25；而#define CRT\_SIZE (CRT\_ROWS \* CRT\_COLS)是显示器屏幕能够容纳的字数，即2000。当crt\_pos大于等于CRT\_SIZE时，说明显示器屏幕已写满，因此将屏幕的内容上移一行，即将第2行至最后1行（也就是第25行）的内容覆盖第1行至倒数第2行（也就是第24行）。接下来，将最后1行的内容用黑色的空格塞满。将空格字符、0x0700进行或操作的目的是让空格的颜色为黑色。最后更新crt\_pos的值。

1. 。
2. He110 World 57616是16进制的0xe110，i的四个字节为r l d \0，改为726c6400。
3. 一个随机的值，具体应该是栈中存储x的位置后面四个字节的值。
4. 按从右到左的方式传递参数。

**练习9：**

movl $(bootstacktop),%esp，这句指令把栈指针的值赋给%esp寄存器。继续往下看，找到bootstack标签，其中.space KSTKSIZE语句申请了大小为KSTKSIZE = 8 \* PGSIZE = 8 \* 4096 字节、初始值全为0的栈空间。再往后定义了bootstacktop标签，可见栈顶位置处于栈的最高地址上，而栈指针指向栈顶，亦即指向栈的最高地址，这也说明栈是由上到下（高地址向低地址）生长的。栈顶的位置我不知道怎么确定的，通过gdb调试观察发现$bootstacktop的值为0xf0110000，这个是虚拟地址，实际的物理地址为0x00110000.

**练习10：**

test\_backtrace函数的调用栈。%esp存储栈顶的位置，%ebp存储调用者栈顶的位置，%eax存储x的值。

将返回地址（call指令的下一条指令的地址）压栈

将ebp, esi, ebx三个寄存器的值压栈，以便退出函数前恢复它们的值

调用cprintf函数打印"entering test\_backtrace x"，其中x为输入参数的值

将输入参数(x-1)压栈，再在栈中分配3个双字，共16字节，以方便清栈

调用test\_backtrace(x-1)

调用cprintf函数打印"leaving test\_backtrace x"，其中x为输入参数的值

每次调用test\_backtrace时共将8个双字压栈：

返回地址

ebp, esi, ebx三个寄存器的值

输入参数(x-1)的值

3个预留双字（与输入参数构成4字节，方便清栈）（将esp寄存器的值减去8）

**练习11：**

代码实现：

1. **int** mon\_backtrace(**int** argc, **char** \*\*argv, **struct** Trapframe \*tf)
2. {
3. uint32\_t \*ebp;
5. ebp = (uint32\_t \*)read\_ebp();
7. cprintf("Stack backtrace:\r\n");
9. **while** (ebp)
10. {
11. cprintf("  ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\r\n",
12. ebp, ebp[1], ebp[2], ebp[3], ebp[4], ebp[5], ebp[6]);
14. ebp = (uint32\_t \*)\*ebp;
15. }
17. **return** 0;
18. }

调用read\_ebp函数来获取当前ebp寄存器的值。ebp寄存器的值实际上是一个指针，指向当前函数的栈帧的底部

前ebp指针存储的恰好是调用者的ebp寄存器的值

内核初始化时会将ebp设置为0，因此当我们检查到ebp为0后就应该结束了。

**练习12：**

代码

1. **int** mon\_backtrace(**int** argc, **char** \*\*argv, **struct** Trapframe \*tf)
2. {
3. uint32\_t \*ebp;
4. **struct** Eipdebuginfo info;
5. **int** result;
7. ebp = (uint32\_t \*)read\_ebp();
9. cprintf("Stack backtrace:\r\n");
11. **while** (ebp)
12. {
13. cprintf("  ebp %08x  eip %08x  args %08x %08x %08x %08x %08x\r\n", ebp, ebp[1], ebp[2], ebp[3], ebp[4], ebp[5], ebp[6]);
15. memset(&info, 0, **sizeof**(**struct** Eipdebuginfo));
17. result = debuginfo\_eip(ebp[1], &info);
18. **if** (0 != result)
19. {
20. cprintf("failed to get debuginfo for eip %x.\r\n", ebp[1]);
21. }
22. **else**
23. {
24. cprintf("\t%s:%d: %.\*s+%u\r\n", info.eip\_file, info.eip\_line, info.eip\_fn\_namelen, info.eip\_fn\_name, ebp[1] - info.eip\_fn\_addr);
25. }
27. ebp = (uint32\_t \*)\*ebp;
28. }
30. **return** 0;
31. }

\_\_STAB\_BEGIN\_\_，\_\_STAB\_END\_\_，\_\_STABSTR\_BEGIN\_\_，\_\_STABSTR\_END\_\_等符号均在kern/kern.ld文件定义，它们分别代表.stab和.stabstr这两个段开始与结束的地址

可见加载内核时符号表也一起加载到内存中了

struct Stab

Symnum是符号索引，换句话说，整个符号表看作一个数组，Symnum是当前符号在数组中的下标

n\_type是符号类型，FUN指函数名，SLINE指在text段中的行号

n\_othr目前没被使用，其值固定为0

n\_desc表示在文件中的行号

n\_value表示地址。特别要注意的是，这里只有FUN类型的符号的地址是绝对地址，SLINE符号的地址是偏移量，其实际地址为函数入口地址加上偏移量。比如第3行的含义是地址f01000b8(=0xf01000a6+0x00000012)对应文件第34行。