#LAB1

- 516030910115
- 吴志文

NOTICE:lab的设计在对应的各个exercise

exercise3

At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?

- 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。

What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?

- 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

How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

- 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。

exercise5

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.)

- 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。

exercise6

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 afterwards!

- 首先将MakeFrag中的0x7c00修改为0x7e00，然后编译
- 0x7c1e处指令，lgdtw 0x7e64 (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，实现实模式转换为保护模式重要的一步)
- 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）

exercise7

Use QEMU and GDB to trace into the JOS kernel and find where the new virtual-to-physical mapping takes effect. Then examine the Global Descriptor Table (GDT) that the code uses to achieve this effect, and make sure you understand what's going on.

- 在这条指令运行之前，地址0x00100000和地址0xf0100000两处存储的内容不一样，但是在这条指令运行之后，存储的内容变为一样。(因为建立了分页机制)

What is the first instruction after the new mapping is established that would fail to work properly if the old mapping were still in place? Comment out or otherwise intentionally break the segmentation setup code in kern/entry.S, trace into it, and see if you were right.

- 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。前访问的逻辑地址超出内存。

exercise8 && exercise9

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. Remember the octal number should begin with '0'.

You need also to add support for the "+" flag, which forces to precede the result with a plus or minus sign (+ or -) even for positive numbers.

- 仿照%d的设计，但是注意是unsign int,并改变base
- exercise9使用一个flag位记录是否有'+'，如果有就putch('+',putdat)

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?
    - console.c export cputchar() for printf.c, printf.c used this function to build a putch() function.
2. Explain the following from console.c：

```
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       }
```

- 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

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?
  - fmt指向的是显示信息的格式字符串，"x %d, y %x, z %d\n"
  - ap会指向所有输入参数的集合
- 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.

4. Run the following code.

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

- He110 World (57616 = 0xe110, 由于x86是little-endian,0x72('r'),0x6c('l'),0x64('d'),0x00('\0'))

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);
```

- x=3 y=-267317640(in my machine,没有参数的y输出的是一个随机值(溢出?))

6. Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change cprintf or its interface so that it would still be possible to pass it a variable number of arguments?

```
#ifndef _STDARG_H
#define _STDARG_H

typedef char *va_list;
```

```
/* Amount of space required in an argument list for an arg of type
TYPE.
TYPE may alternatively be an expression whose type is used.  */

#define __va_rounded_size(TYPE)  \
(((sizeof (TYPE) + sizeof (int) - 1) / sizeof (int)) * sizeof (int))

#ifndef __sparc__
#define va_start(AP, LASTARG)
\
(AP = ((char *) &(LASTARG) + __va_rounded_size (LASTARG)))
#else
#define va_start(AP, LASTARG)
\
(__builtin_saveregs (),                                         \
AP = ((char *) &(LASTARG) + __va_rounded_size (LASTARG)))
#endif

void va_end (va_list);          /* Defined in gnulib */
#define va_end(AP)

#define va_arg(AP, TYPE)
\
(AP += __va_rounded_size (TYPE),
\
*((TYPE *) (AP - __va_rounded_size (TYPE))))

#endif /* _STDARG_H */
```

  - 编译器从右往左向栈内压入参数,后压栈的在内存的低地址，如果要保证从左往右的取出每个变量的话)

exercise10 && exercise11

- 支持%n传入char *,把当前输出的位置，写回传来的变量
  - putdat的用法其实是一个int*,正好可以表示输出的位置
  - 根据va_arg(ap,char *)可以判断是否为传入的参数是否为nullptr,如果putdat超出范围了，就报overflow错误，这里和exercise16有关联，如果范围设置不正确后面的exercise就会错误。
- 如果遇见'-'，则说明要左对齐，之后用' '填充。先将num翻转，比如10进制1234翻转为4321。因为printnum函数使用，再从低位打印，填充' '，得到[1234]（注意将判断逻辑放在原代码之前，不影响其他情况

exercise14 && exercise15

- 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。
- 通过jos作者给的数据结构debuginfo_eip(大部分已经完成，只需在kdebug.c中fufil部分代码),将mon_backtrace()按给定的格式打印。

exercise16

- 用cprintf和之前实现的%n来将返回地址写为do_overflow,原理和icslab buffer overflow类似