Jos Lab 1: Booting a PC

Li Xinyu 515030910292

Design of Lab1

> Lab1包含5个文件夹：boot文件夹内各个文件为bootloader；conf文件夹内为参数配置文件；inc(即include)文件夹内是一些被其他部分引用头文件；kern文件夹内即为kernel的代码；lib文件夹内为一些辅助函数的实现代码。另外编译之后会生成文件夹obj，包含可执行文件和汇编
> 当在命令行输入make qemu时，qemu虚拟机启动然后在0xF0000处载入BIOS，执行的第一条指令为`0xffff0: ljmp $0xf000,$0xe05b`，之后BIOS对一些重要的硬件设备如VGA Display进行初始化。然后BIOS从硬盘中读取bootloader将其加载到内存中，最后BIOS跳转到0x7C00移交控制开始执行bootloader。
> Bootloader执行的第一条指令为`cli`即关闭中断，然后进行其他的初始化操作：打开A20地址线，加载GDT，然后从real mode切换到protected mode并初始化寄存器DS、ES、FS、GS和SS，最后调用bootmain加载内核。Bootmain首先读取硬盘的第一个page，根据其ELF header的索引找到ELF中各个segment的位置，然后将其读取加载到内存中，最后跳转到ELF header的entry point的物理地址进入内核。
> 内核首先执行entry.S中的代码，利用entry_pgdir.c中的代码建立kernel虚拟内存的页表，将页表物理地址加载到`%cr3`,然后切换到页模式的寻址方式。之后初始化内核的栈空间然后调用init.c中的i386_init函数。i386_init初始化终端，然后调用一些cprintf函数和test_backtrace函数。这里的一些函数调用主要是用来查看之后的一些exercises中所需要完成的功能是否正确输出。test_backtrace用来调用mon_trace进行递归回溯栈上的函数调用信息并进行输出。在完成以上工作后，进入死循环调用monitor等待用户在终端上的输入。至此系统启动完成。

Part 1: PC Bootstrap

> **Exercise 2**
> BIOS设置中断描述表， 初始化一些重要的硬件设备，然后从启动设备如硬盘上读取bootloader并转移控制。

Part 2: The Boot Loader

> **Exercise 3**

- 在`movl %eax, %cr0`之后开启保护模式，处理器开始执行32位代码`ljmp $PROT_MODE_CSEG, $protcseg`真正地从16位切换到32位
- Bootloader执行的最后一条语句是`((void (*)(void)) (ELFHDR->e_entry))()`其汇编指令为`ff 15 18 00 01 00 call *0x10018`；
  Kernel执行的第一条指令为`movw $0x1234,0x472 #warm boot`
- Bootloader首先读取磁盘的第一个页得到ELFHEADER，通过ELFHEADER的e_phnum得知需要加载的扇区数量

Loading the Kernel

Exercise 4
第一行输出中a为栈分配的一块16字节区域的起始地址，地址较高；b是堆上分配的一块16字节区域的起始地址，地址比a低；c是一个未初始化的指针，可能是一个未知随机内容或者nil；
第二行输出中因为c指向了a所指区域，所以c[0]=200使得数组a中的第一个元素值为200；
第三、四行输出结果的原因与第二行相同，而3[c]这种写法等同于c[3]！；
第五行输出中因为`c=(int *)((char*)c + 1)`先将c视为char指针，+1使其移动8位而非32位，之后再转成int指针使其指向a[1]的第9位到a[2]的第8位之间的4个字节，所以当使得`*c=500`时，更改了这四个字节的内容，即改变了a[1]和a[2]的值；
第六行输出中`b = (int *) a + 1`即b指向比a高4个字节的地址，而`c = (int *) ((char *) a + 1)`则将c指向比a高一个字节的地址(理由如第五行)。

-

Exercise 5
由于0x00100000为kernel的代码区域，进入bootloader前和进入kernel前这两个时间点上0x00100000所在地址的8个字节内区域中的内容不同，是因为在bootloader之前并没有将内核代码加载到内存中，之后bootoader的bootmain将内核代码载入内存因此导致不同。

Link vs. Load Address

Exercise 6
将Makefrag文件中`$(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o $@.out$^`的0x7c00改为0x7c04，然后重新make发现boot.asm文件中指令的地址均+4，即原先第一条指令地址变为0x7C04。在使用gdb调试时，`b *0x7c00`发现仍然停在指令`[ 0:7c00] => 0x7c00: cli`，且之后的指令均为booloader中紧接着的指令，说明修改link-address并未改变load-address。但是在执行到`[0:7c2d] => 0x7c2d: ljmp $0x8,$0x7c36`时，其下一跳执行的指令却变成了`[f000:e05b] 0xfe05b: cmpl $0x0,%cs:0x70b8`,说明修改link-address影响了与地址相关的指令ljmp。

Part 3: The Kernel

Using segmentation to work around position dependence

> **Exercise 7**
> 在执行完`0x10002d: jmp *%eax`之后的下一条指令地址变为0xf10002f，说明新的virtual-to-physical映射已建立。如果将entry.S中的这几条对应指令注释掉，则发现执行完`0x10001a: jmp *%eax`后，结果如下图

```
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) b *0x10000c
Breakpoint 1 at 0x10000c
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    movw   $0x1234,0x472

Breakpoint 1, 0x0010000c in ?? ()
(gdb) si
=> 0x100015:    mov    $0xf010001c,%eax
0x00100015 in ?? ()
(gdb) si
=> 0x10001a:    jmp    *%eax
0x0010001a in ?? ()
(gdb) si
=> 0xf010001c <relocated>:      add    %al,(%eax)
relocated () at kern/entry.S:73
73              movl    $0x0,%ebp                       # nuke frame pointer
(gdb) si
Remote connection closed
(gdb)
```

连接
关闭的原因是qemu窗口关闭停止运行，报错如下

```
qemu-system-i386: Trying to execute code outside RAM or ROM at 0xf010001c
This usually means one of the following happened:

(1) You told QEMU to execute a kernel for the wrong machine type, and it crashed
 on startup (eg trying to run a raspberry pi kernel on a versatilepb QEMU machin
e)
(2) You didn't give QEMU a kernel or BIOS filename at all, and QEMU executed a R
OM full of no-op instructions until it fell off the end
(3) Your guest kernel has a bug and crashed by jumping off into nowhere

This is almost always one of the first two, so check your command line and that
you are using the right type of kernel for this machine.
If you think option (3) is likely then you can try debugging your guest with the
 -d debug options; in particular -d guest_errors will cause the log to include a
 dump of the guest register state at this point.

Execution cannot continue; stopping here.

GNUmakefile:148: recipe for target 'qemu-gdb' failed
make: *** [qemu-gdb] Error 1
```

说明新的映射未建立导致内核指令无法继续执行，运行出错。

Formatted Printing to the Console

> **Exercise 8**
> 为输出8进制数字，在函数`vprintfmt`中`case 'o':`语句中参考`case 'u':`和`case 'x':`,从ap中获取num–`num = getuint(&ap, lflag);`并将base设置为8。

-

Exercise 9
为了在正数时输出”+”，在格式串%后读取到+时设置flag为true，然后在`case 'd':`语句下根据flag确定是否输出”+”。

-

Qusetions:

1。console.c提供的函数有cputchar，getchar，iscons；printf.c调用console.c中的cputchar函数将字符打印到终端显示窗口。

-

2。这段代码用于当屏幕输出占满一行时换行。

-

3。

- `fmt` 指向格式字符串，即 `cprintf` 中第一个参数； `ap` 指向该函数的其他参数
- 使用gdb设置断点获取以下信息：
  vcprintf (fmt=0xf0101eb6 “x %d, y %x, z %d\n”, ap=0xf010ffb4 “\001”)
  cons_putc (c=120)
  cons_putc (c=32)
  cons_putc (c=49)
  cons_putc (c=44)
  cons_putc (c=32)
  cons_putc (c=121)
  cons_putc (c=32)
  cons_putc (c=51)
  cons_putc (c=44)
  cons_putc (c=32)
  cons_putc (c=122)
  cons_putc (c=32)
  cons_putc (c=52)
  cons_putc (c=10)
  未能获知va_arg的调用信息。。。

-

4。输出为 `He110 World` 。57616表示为十六进制即0xe110，数字的写入和读取都是按小端法，所以前半部分为 `He110` ； 0x00646c72在内存中用小端法存储即为如

下所示。字符串从低地址读到高地址，所以后半部分为 `World` 。
+———+ 高地址
+ 0x00 + <= `\0`
+ 0x64 + <= `d`
+ 0x6c + <= `l`
+ 0x72 + <= `r`
+———+ 低地址

-

5。 `y=` 之后会输出一个无法预料到的值，其内容等同于函数 `cprintf` 在栈上空间 `%esp+12` 所指地址4字节区域内的内容。

-

6。假设GCC改变参数压栈顺序，那么可以在 `cprintf` 的最后传入一个参数表示传入的参数个数，那么这个值将被第一个读到并用来确定之后要从栈上读取几个参数。

-

Exercise 10
修改函数 `printnum` ,将 `printnum` 原有代码移到helper函数中，并添加参数 `int* cnt` ，用于记录已经输出的长度，每次输出时 `++(*cnt)` ，在调用helper函数输出完数字后若 `*cnt` 小于 `width` 且 `padc=='-'` 则输出 `' '` 。

The Stack

Exercise 11
初始化栈的指令为 `movl $0x0,%ebp #nuke frame pointer` 和 `movl $(bootstacktop),%esp` ，根据kernel.asm文件中
`movl $(bootstacktop),%esp`
`f0100021: bc 00 00 11 f0 mov $0xf0110000,%esp` 可知kernel初始化地址在0xf0110000即%esp所指的地址。

-

Exercise 12
test_backtrace一次调用使用的栈空间为32bytes。在调用 `cprintf` 之前，压入 `ebp` 、 `ebx` ,将 `esp` 下移12bytes，然后压入 `ebx` 、静态字符串地址， `call` 指令压入返回地址，至此使用了32bytes空间。调用 `cprintf` 之后 `esp` 指向返回地址上面4个字节，在

> 执行递归之前，在原空间变化为-16+12，压入 `eax` 和返回地址+8，正好又回到了32bytes。

-

> **Exercise 13 & 14**
> 修改函数 `mon_backtrace` ，首先读取 `edp` ，然后读取栈上 `edp` 上方的 `eip` 以及5个四字节参数。然后利用 `debuginfo_eip()` 根据读取的 `eip` 找到相关的调用函数的信息，并将其输出。由于kernel初始化时将 `edp` 设置为了0，所以判断循环结束条件就是 `edp == 0` 。
> 修改 `debuginfo_eip` ，添加对 `stab_binsearch` 的调用从而找到调用函*return instruction pointer*的地址在对应函数中的偏移量。由于需要查询代码段的行号，查看 `stab.h` 文件得知 `#define N_SLINE 0x44 // text segment line number` 因此调用 `stab_binsearch` 时类型参数为N_SLINE。

-

> **Exercise 15**
> 添加函数 `mon_time` ，查看intel手册得知RDTSC将时间戳计数器(64位)加载到EDX:EAX中，EDX存放高32位，EAX存放低32位。因此函数执行前后各调用一次”rdtsc”指令并读取寄存器 `edx` 和 `eax` 的值，将两次的计数器的值相减即为运行的cycles。