lab1实验报告

1. 环境配置

首先是Qemu硬件模拟器安装

安装前可以进行检测本虚拟机是否已有qemu

检测：命令行 q (按TAB键)

若已存在qemu则跳过安装步骤

若没有：

安装 命令行：sudo apt-get install qemu-system（安装文件较大 要等待）

完成后再进行检测是否已有qmeu硬件模拟器

此时已安装好实验环境，可以进行接下来的实验了。

1. 练习1

理解通过make生成可执行文件的过程及原理

在命令行中输入：make V=

可以分析ucore.img的生成

1. 在makefile文件中，关于ucore.img的代码为：

----------------------------------------------------------

$(UCOREIMG): $(kernel) $(bootblock)====》由此需要kernel与booklock

$(V)dd if=/dev/zero of=$@ count=10000

$(V)dd if=$(bootblock) of=$@ conv=notrunc

$(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc

----------------------------------------------------------

1. kernel生成代码为：

----------------------------------------------------------

$(kernel): tools/kernel.ld

$(kernel): $(KOBJS)

@echo + ld $@

$(V)$(LD) $(LDFLAGS) -T tools/kernel.ld -o $@ $(KOBJS)

@$(OBJDUMP) -S $@ > $(call asmfile,kernel)

@$(OBJDUMP) -t $@ | $(SED) '1,/SYMBOL TABLE/d; s/ .\* / /; \

/^$$/d' > $(call symfile,kernel)

----------------------------------------------------------

1. booklock生成代码为：

----------------------------------------------------------

$(bootblock): $(call toobj,$(bootfiles)) | $(call totarget,sign)

@echo + ld $@

$(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 $^ \

-o $(call toobj,bootblock)

@$(OBJDUMP) -S $(call objfile,bootblock) > \

$(call asmfile,bootblock)

@$(OBJCOPY) -S -O binary $(call objfile,bootblock) \

$(call outfile,bootblock)

@$(call totarget,sign) $(call outfile,bootblock) $(bootblock)

-----------------------------------------------------------

1. 练习2

本练习的目的是使用qemu动态调试，使我们理解计算机加电后 BIOS执行过程

1.修改目录下labcodes/lab1/tools/gdbinit的内容为:

set architecture i8086

target remote :1234

---------------------------------------------------------------------

2.cd到 lab1目录下，执行

命令行命令：make debug

---------------------------------------------------------------------

3.在看到gdb的调试界面(gdb)后，在gdb调试界面下执行如下命令

命令行命令：si

即可单步跟踪BIOS了。

---------------------------------------------------------------------

4.在gdb界面下，可通过如下命令来看BIOS的代码

gdb调试界面: x /2i $pc //显示当前eip处的汇编指令

其中2i是打印两条汇编指令，可以根据需要来自行修改其数值。

查看下一条汇编指令可以使用命令：next

1. 练习3

本练习是在分析bootloader 进入保护模式的过程。

从`%cs=0 $pc=0x7c00`，进入后：

1.首先清理环境：包括将flag置0和将段寄存器置0

-----------------------------------------------

.code16

cli

cld

xorw %ax, %ax

movw %ax, %ds

movw %ax, %es

movw %ax, %ss

-----------------------------------------------

2.开启A20：通过将键盘控制器上的A20线置于高电位，全部32条地址线用，

可以访问4G的内存空间。

------------------------------------------------

seta20.1: # 等待8042键盘控制器不忙

inb $0x64, %al #

testb $0x2, %al #

jnz seta20.1 #

movb $0xd1, %al # 发送写8042输出端口的指令

outb %al, $0x64 #

seta20.1: # 等待8042键盘控制器不忙

inb $0x64, %al #

testb $0x2, %al #

jnz seta20.1 #

movb $0xdf, %al # 打开A20

outb %al, $0x60 #

-------------------------------------------

3.初始化GDT表：一个简单的GDT表和其描述符已经静态储存在引导区中，载入即可

--------------------------

lgdt gdtdesc

--------------------------

4.进入保护模式：通过将cr0寄存器PE位置1便开启了保护模式

--------------------------------

movl %cr0, %eax

orl $CR0\_PE\_ON, %eax

movl %eax, %cr0

--------------------------------

5.通过长跳转更新cs的基地址

--------------------------------

ljmp $PROT\_MODE\_CSEG, $protcseg

.code32

protcseg:

--------------------------------

6.设置段寄存器，并建立堆栈

--------------------------------

movw $PROT\_MODE\_DSEG, %ax

movw %ax, %ds

movw %ax, %es

movw %ax, %fs

movw %ax, %gs

movw %ax, %ss

movl $0x0, %ebp

movl $start, %esp

--------------------------------

7.转到保护模式完成，进入boot主方法

--------------------------------

call bootmain

--------------------------------

1. 练习4

本练习旨在分析bootloader加载ELF格式的OS的过程。

1.

首先看readsect函数，

`readsect`从设备的第secno扇区读取数据到dst位置

--------------------------------------------------------

static void

readsect(void \*dst, uint32\_t secno) {

waitdisk();

outb(0x1F2, 1); // 设置读取扇区的数目为1

outb(0x1F3, secno & 0xFF);

outb(0x1F4, (secno >> 8) & 0xFF);

outb(0x1F5, (secno >> 16) & 0xFF);

outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);

// 上面四条指令联合制定了扇区号

// 在这4个字节线联合构成的32位参数中

// 29-31位强制设为1

// 28位(=0)表示访问"Disk 0"

// 0-27位是28位的偏移量

outb(0x1F7, 0x20); // 0x20命令，读取扇区

waitdisk();

insl(0x1F0, dst, SECTSIZE / 4); // 读取到dst位置，

// 幻数4因为这里以DW为单位

}

--------------------------------------------------------

2.

readseg简单包装了readsect，可以从设备读取任意长度的内容。

--------------------------------------------------------

static void

readseg(uintptr\_t va, uint32\_t count, uint32\_t offset) {

uintptr\_t end\_va = va + count;

va -= offset % SECTSIZE;

uint32\_t secno = (offset / SECTSIZE) + 1;

// 加1因为0扇区被引导占用

// ELF文件从1扇区开始

for (; va < end\_va; va += SECTSIZE, secno ++) {

readsect((void \*)va, secno);

}

}

---------------------------------------------------------

3.

在bootmain函数中，

---------------------------------------------------------

void

bootmain(void) {

// 首先读取ELF的头部

readseg((uintptr\_t)ELFHDR, SECTSIZE \* 8, 0);

// 通过储存在头部的幻数判断是否是合法的ELF文件

if (ELFHDR->e\_magic != ELF\_MAGIC) {

goto bad;

}

struct proghdr \*ph, \*eph;

// ELF头部有描述ELF文件应加载到内存什么位置的描述表，

// 先将描述表的头地址存在ph

ph = (struct proghdr \*)((uintptr\_t)ELFHDR + ELFHDR->e\_phoff);

eph = ph + ELFHDR->e\_phnum;

// 按照描述表将ELF文件中数据载入内存

for (; ph < eph; ph ++) {

readseg(ph->p\_va & 0xFFFFFF, ph->p\_memsz, ph->p\_offset);

}

// ELF文件0x1000位置后面的0xd1ec比特被载入内存0x00100000

// ELF文件0xf000位置后面的0x1d20比特被载入内存0x0010e000

// 根据ELF头部储存的入口信息，找到内核的入口

((void (\*)(void))(ELFHDR->e\_entry & 0xFFFFFF))();

bad:

outw(0x8A00, 0x8A00);

outw(0x8A00, 0x8E00);

while (1);

}

----------------------------------------------------------------

1. 练习5

本练习是实现函数调用堆栈跟踪函数

ss:ebp指向的堆栈位置储存着caller的ebp，以此为线索可以得到所有使用堆栈的函数ebp。

ss:ebp+4指向caller调用时的eip，ss:ebp+8等是（可能的）参数。

输出中，堆栈最深一层为

-------------------------------------------------------------------

ebp:0x00007bf8 eip:0x00007d68 \

args:0x00000000 0x00000000 0x00000000 0x00007c4f

<unknow>: -- 0x00007d67 --

-------------------------------------------------------------------

其对应的是第一个使用堆栈的函数，bootmain.c中的bootmain。

bootloader设置的堆栈从0x7c00开始，使用"call bootmain"转入bootmain函数。

call指令压栈，所以bootmain中ebp为0x7bf8。

1. 练习6

完善中断初始化和处理

1.中断向量表中一个表项占多少字节？其中哪几位代表中断处理代码的入口？

答：

中断向量表一个表项占用8字节，其中2-3字节是段选择字，0-1字节和6-7字节拼成位移，两者联合便是中断处理程序的入口地址。

2.请编程完善kern/trap/trap.c中对中断向量表进行初始化的函数idt\_init。

答：

在 idt\_init 函数中,依次对所有中断入口进行初始化。使用 mmu.h 中的 SETGATE 宏,填充 idt 数组内容。注意除了系统调用中断(T\_SYSCALL)以外,其它中断均使用中断门描述符,权限为内核态权限;而系统调用中断使用 异常,权限为陷阱门描述符。每个 中断的入口由tools/vectors.c 生成,使用 trap.c 中声明的 vectors 数组即可。

填充的代码为

--------------------------------------------------------------------

void idt\_init(void) {

extern uintptr\_t \_\_vectors[]; //声明\_\_vertors[],其中存放着中断服务程序的入口地址

int i;

for(i=0;i<256;i++) {

SETGATE(idt[i],0,GD\_KTEXT,\_\_vectors[i],DPL\_KERNEL);

}

SETGATE(idt[T\_SWITCH\_TOK],0,GD\_KTEXT,\_\_vectors[T\_SWITCH\_TOK],DPL\_USER); //填充中断描述符表IDT

lidt(&idt\_pd); //使用lidt指令加载中断描述符表

}

--------------------------------------------------------------------

3. 请编程完善trap.c中的中断处理函数trap，在对时钟中断进行处理的部分填写trap函数

1. 小结
2. 内容很多，掌握的也很不熟练，大部分代码时参考的教程，理解还不是太深刻，但是通过源代码审计对引导工作有了一定的认识。
3. 编写代码能力不足，在实验中，很难去独立完成代码，需不断地积累学习。
4. 对BIOS有了初步的理解，但是对其详细原理还是有些不懂，要在接下来的学习中，继续加强对此方面相关知识的学习和掌握。
5. 实验原理以及分析解释部分参考了教程以及答案，虽然对这些知识有了初步的理解，但仍需要加强这方面能力的提高。