1. **实验环境配置**

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

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

1. **lab1 练习1：理解通过make生成执行文件的过程。**

由于make生成文件的过程只会显示输出，若想详细了解其执行了什么命令可在命令行中执行：make“V=”

①在makefile文件中，关于ucore.img的代码为：

1. $(UCOREIMG): $(kernel) $(bootblock)====》由此需要kernel与booklock
2. $(V)dd if=/dev/zero of=$@ count=10000
3. $(V)dd if=$(bootblock) of=$@ conv=notrunc
4. $(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc

②kernel生成代码：

$(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

$(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. **Lab1 练习2：使用qemu动态调试，理解计算机加电后 BIOS执行过程**

1 添加labcodes/lab1/tools/gdbinit的内容为:

target remote :1234

2 lab1目录下，执行：make debug

3 在看到gdb的调试界面后，在gdb调试界面下执行如下命令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

显示当前eip处的汇编指令：x /2i $pc

1. **lab1练习3：分析bootloader进入保护模式的过程**

①初始化环境

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。

③加载GDT的基地址 lgdt gdtdesc

④进入bootmain函数

此时转到保护模式完成：

1)将cr0寄存器PE位置1便开启了保护模式

ljmp $PROT\_MODE\_CSEG, $protcseg

.code32

protcseg:

2) 进入bootmain函数

call bootmain

**Lab1练习4：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);

}

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

bad:

outw(0x8A00, 0x8A00);

outw(0x8A00, 0x8E00);

while (1);

}

1. **Lab1练习5：完成kdebug.c中函数print\_stackframe的实现**

可以通过函数>print\_stackframe来跟踪函数调用堆栈中记录的返回地址。

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. **Lab1练习6：完成中断初始化和处理**
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。

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

}

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

（二）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 函数中处理时钟中断的部分,使操作系统每遇到 100 次时钟中断后,调用 print\_ticks 子程序,向屏幕上打印一行文字”100 ticks”。

代码：

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

Case IRQ\_OFFSET + IRQ\_TIMER:

Ticks ++;

If(ticks % TICK\_NUM == 0)

{

Printf\_ticks();

}

Break;

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