## Ucore实验lab1

### 练习一：理解通过make生成执行文件的过程。

#### 在Makefile中生成ucore.img的代码是：

$(UCOREIMG): $(kernel) $(bootblock) $(V)dd if=/dev/zero of=$@ count=10000 $(V)dd if=$(bootblock) of=$@ conv=notrunc $(V)dd if=$(kernel) of=$@ seek=1 conv=notrunc

#### 执行Makefile时对应的输出为：

dd if=/dev/zero of=bin/ucore.img count=1000010000+0 records in10000+0 records out5120000 bytes (3.8 MB) copied, 0.0784515 s, 55.8 MB/sdd if=bin/bootblock of=bin/ucore.img conv=notrunc1+0 records in1+0 records out512 bytes (512 B) copied, 0.0002006 s, 5.6 MB/sdd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc138+1 records in138+1 records out71845 bytes (72 kB) copied, 0.0005683 s, 150 MB/s

##### 从Makefile中可以看出要生成ucore.img首先要拥有kernel和bootblock两个可执行文件所以继续在Makefile中找到kernel的相关代码段

kernel = $(call totarget,kernel) $(kernel): tools/kernel.ldw$(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) $(call create\_target,kernel)

#### 查看文件中的文件得出，生成kernel需要以下文件：

kernel.ld init.o readline.o stdio.o kdebug.o kmonitor.o panic.o clock.o console.o intr.o picirq.o trap.o trapentry.o vectors.o pmm.o printfmt.o string.o

在Makefile中bootblock的相关代码：# create bootblockbootfiles = $(call listf\_cc,boot)$(foreach f,$(bootfiles),$(call cc\_compile,$(f),$(CC),$(CFLAGS) -Os -nostdinc)) bootblock = $(call totarget,bootblock) $(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) $(call create\_target,bootblock)

#### 生成ucore.img过程中产生的信息：

dd if=/dev/zero of=bin/ucore.img count=1000010000+0 records in10000+0 records out5120000 bytes (5.1 MB) copied, 0.0540315 s, 94.8 MB/sdd if=bin/bootblock of=bin/ucore.img conv=notrunc1+0 records in1+0 records out512 bytes (512 B) copied, 0.000110706 s, 4.6 MB/sdd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc138+1 records in138+1 records out70775 bytes (71 kB) copied, 0.000472623 s, 150 MB/s

### 练习二使用qemu执行并调试lab1中的软件

#### 1从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行。

修改

lab1/tools/gdbinitset architecture i8086target remote :1234

执行

make debug

来查看BIOS的代码，得到结果。

#### 2在初始化位置0x7c00设置实地址断点,测试断点正常。

在gdbinit 中添加指令：set architecture i8086b \*0x7c00cx/5i $pcset architecture i386

然后运行Make debug可以得到断点正常。

#### 3从0x7c00开始跟踪代码运行,将单步跟踪反汇编得到的代码与bootasm.S和 bootblock.asm进行比较。

​ 在0x7c00处break，然后使用si和 x/i $pc 指令一行一行的跟踪，将得到的反汇编代码为：0x00007c01 in ?? ()(gdb) x/i $pc=> 0x7c01: cld (gdb) si0x00007c02 in ?? ()(gdb) x/i $pc=> 0x7c02: xor %eax,%eax(gdb) si0x00007c04 in ?? ()(gdb) x/i $pc=> 0x7c04: mov %eax,%ds(gdb)

bootblock.asm 中的代码为：.

code16 # Assemble for 16-bit mode cli # Disable interrupts 7c00: fa cli cld # String operations increment 7c01: fc cld # Set up the important data segment registers (DS, ES, SS). xorw %ax, %ax # Segment number zero 7c02: 31 c0 xor %eax,%eax movw %ax, %ds # -> Data Segment 7c04: 8e d8 mov %eax,%ds movw %ax, %es # -> Extra Segment 7c06: 8e c0 mov %eax,%es movw %ax, %ss # -> Stack Segment 7c08: 8e d0 mov %eax,%ss

###### bootasm.s中的代码为：

.globl startstart:.code16 # Assemble for 16-bit mode cli # Disable interrupts cld # String operations increment # Set up the important data segment registers (DS, ES, SS). xorw %ax, %ax # Segment number zero movw %ax, %ds # -> Data Segment movw %ax, %es # -> Extra Segment movw %ax, %ss # -> Stack Segment

###### 4自己找一个bootloader或内核中的代码位置，设置断点并进行测试

###### 在init.c中会对gdt进行初始化，在初始化过程中，调用了gdt\_init()这个函数，我选择此处进行测试。已经成功在gdt\_init处停止，然后使用x/i $pc指令查询正在执行的汇编代码，与原文件中的c代码对照，测试结束。 \*\*练习三分析bootloader进入保护模式的过程。

###### \*\*从bootasm.s查看代码，并分析过程首先禁止中断，并且修改控制方向标志寄存器DF=0，使得内存地址从低到高增加，然后对实模式下的段寄存器进行初始化

###### xorw %ax, %ax 将ax寄存器置为0，然后，对ds（数据段寄存器），es（额外寄存器），ss（栈寄存器）

seta20.1: inb $0x64, %al

testb $0x2, %al jnz seta20.1

movb $0xd1, %al # 0xd1 -> port 0x64

outb %al, $0x64 seta20.2: inb $0x64, %al

testb $0x2, %al

jnz seta20.2

movb $0xdf, %al # 0xdf -> port 0x60

outb %al, $0x60

###### 接下来初始化GDT表，GDT表已经在数据区声明

# Bootstrap GDT .p2align 2 # force 4 byte alignmentgdt: SEG\_NULLASM # null seg SEG\_ASM(STA\_X|STA\_R, 0x0, 0xffffffff) # code seg for bootloader and kernel SEG\_ASM(STA\_W, 0x0, 0xffffffff) # data seg for bootloader and kernel gdtdesc: .word 0x17 # sizeof(gdt) - 1 .long gdt # address gdt

###### 然后将cr0置为1开启保护模式,设置保护模式下的段寄存器

movw $PROT\_MODE\_DSEG, %ax # Our data segment selector movw %ax, %ds # -> DS: Data Segment movw %ax, %es # -> ES: Extra Segment movw %ax, %fs # -> FS movw %ax, %gs # -> GS movw %ax, %ss # -> SS: Stack Segment PROT\_MODE\_DSEG为0x10

###### 建立堆栈，并且调用bootmain.c

movl $0x0, %ebp

movl $start, %esp

call bootmain

###### 栈的地址是从0到0x7c00

### 练习四分析bootloader加载ELF格式的OS的过程

#### bootloader首先读取ELF头表

// read the 1st page off disk

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

###### ELFHDR是前面定义的一个临时空间，而uintptr\_t是在defs.h中定义的：

#define ELFHDR ((struct elfhdr \*)0x10000) // scratch space

所以一层层推导下来，uintptr\_t就是unsigned int。

###### 这里是为了把ELF的头表读进去，传入readseg的三个参数分别表示，起始位置，大小，和偏移。大小是SECTSIZE8，SETCSIZE的大小是512：

#define SECTSIZE 512

###### 这里读入了5128个位，然而实际并不需要这么大，亲测只要一个SECTSIZE也就是512就足够，所以猜测可能是为了以后的扩展，所以在这里定义了这么大。当头表读入完成后，就通过e\_magic检测是否是合法的ELF文件：

// is this a valid ELF? if (ELFHDR->e\_magic != ELF\_MAGIC) { goto bad; }

如不是就会跳入到bad执行指令，如果是就继续往下执行。根据ELF头表中的描述，将ELF文件加载到内存中的相应位置保存到ph中;

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

将结束位置加载到eph中,然后就将ELF文件中的数据通过循环加载到内存中，加载完成后根据ELF头表中存储的入口信息，找到内核的入口：

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

#### 5实现函数调用堆栈跟踪函数

###### 代码：

uint32\_t ebp = read\_ebp(); uint32\_t eip = read\_eip(); int i,j; for(i = 0; ebp!=0 && i < STACKFRAME\_DEPTH; i++) { cprintf("ebp:0x%08x eip:0x%08x ",ebp, eip); cprintf("args"); uint32\_t \*args = (uint32\_t \*)ebp + 2; for(j = 0; j < 4; j++) cprintf("0x%08x ",args[j]); cprintf("\n"); print\_debuginfo(eip-1); eip = \*((uint32\_t \*)ebp+1); ebp = \*((uint32\_t \*)ebp+0); }

### 练习六完善中断初始化和处理

#### 1中断描述符表（也可简称为保护模式下的中断向量表）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？

中断描述符表中一个表项占8个字节，其中015和4863分别为offset的低16位和高16位，16~31位是段选择子，通过段选择子得到段基址，再加上段内偏移量就可以得到中断处理代码的入口。

#### 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。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。

###### 代码如下：

extern uintptr\_t \_\_vectors[]; int i = 0; for(i = 0; i<sizeof(idt) / sizeof(struct gatedesc);i++) { SETGATE(idt[i], 0,KERNEL\_CS, \_\_vectors[i], DPL\_KERNEL); } SETGATE(idt[T\_SYSCALL], 0, KERNEL\_CS, \_\_vectors[T\_SYSCALL], DPL\_USER); lidt(&idt\_pd);

###### 