|  |  |  |  |
| --- | --- | --- | --- |
| **姓名** | 李昊阳 | **学号** | 14 |
| **实验题目** | Lab1-启动操作系统 | | |
| **实验内容** | （练习1）了解ucore的“项目组成”。  使用ls查看ucore中lab1的项目组成    （练习1）会使用make编译文件，了解Makefile中的主要组成，实验报告中要有简单介绍。查看使用make与make “V=”所生成信息的不同。 根据make编译信息，简要说明gcc编译器是如何一步步生成ucore可执行文件的。sign.c的作用是什么。  在文件夹中打开makefile查看    可以看出Makefile通过   1. 为每一个源文件（.c和.S文件）产生一个描述其依赖关系的makefile文件，以.d为后缀。 2. 编译源文件，只生成目标文件但不链接。包括由bootasm.S、bootmain.c、sign.c分别生成bootasm.o、bootmain.o、sign.o。 3. 由目标文件sign.o生成可执行文件sign，生成sign工具（生成引导区内容） 4. 链接bootasm.o、bootmain.o生成bootblock.o，设定程序入口为start，地址为0x7C00。 5. 将bootblock.o内容进行反汇编，写入文件bootblock.asm。 6. 将目标文件bootblock.out全部内容拷贝到bootblock.o中，并转换为raw二进制（raw binary）格式。 7. 使用sign工具生成引导区内容装载bootblock.out，生成bin目录下的bootblock。 8. 生成ucore.img。   等8个步骤生成ucore.img  在lab1文件夹中打开终端，输入make      首先输入make clean，再输入make “V=”，否则会是重复的操作      当使用make V=时，会看到更加详细的操作过程，最后生成.o文件  编译器gcc预处理，生成预编译文件（.i文件）：gcc –E main.c –o main.i  编译，生成汇编代码（.s文件）：gcc –S main.i –o main.s  汇编，生成目标文件（.o文件）：gcc –c main.s –o main.o  链接，生成可执行文件（executable文件）：gcc main.o –o main  Sign工具是处理bootblock.out，生成bootblock  （练习2）学会使用qemu与gdb协作进行调试ucore代码；从CPU加电后执行的第一条指令开始，单步跟踪BIOS的执行。在初始化位置0x7c00设置实地址断点,测试断点正常。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。自己找一个bootloader或内核中的代码位置，设置断点并进行测试。（截图并简要说明实验过程）  由于实验环境的问题（缺少qemu），之后的实验在实验楼在线平台完成：    实验楼网址：<https://www.shiyanlou.com>  修改tools里面的gdbinit和Makefile里面的debug部分，然后执行make debug      查看BIOS的代码    再次修改gdbinit，在初始化位置0x7c00设置实地址断点,测试断点正常    （练习3）了解如何在bootloader中完成由实模式到保护模式的转换的。（要有过程或对应代码的截图）  从`%cs=0 $pc=0x7c00`，进入后  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  ```  开启A20：通过将键盘控制器上的A20线置于高电位，全部32条地址线可用，  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 #  ```  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  ```  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  ```  通过长跳转更新cs的基地址  ```  ljmp $PROT\_MODE\_CSEG, $protcseg  .code32  protcseg:  ```  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  ```  转到保护模式完成，进入boot主方法  ```  call bootmain  ```  （练习4）根据代码，通过阅读bootmain.c，了解bootloader如何加载ELF文件。（需要截图，并简要说明）  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  }  ```  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);  }  }  ```  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);  }  ```  （练习5）要求完成函数kern/debug/kdebug.c::print\_stackframe的实现.完成kdebug.c中函数print\_stackframe的实现。（需要make qemu 后的结果等截图，简要说明打印信息，说明调用关系。）    在kern/debug/kdebug.c中添加上图所示函数，实现print\_stackframe的功能  之后在lab1文件中打开终端，输入make qemu      输出与原表大致一致  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  （练习6）从代码找出中断描述符表（也可简称为保护模式下的中断向量表）的定义，并简要说明中断描述符表中一个表项占多少字节？各分别表示什么？其中哪几位代表中断处理代码的入口？（截图并简要说明）    中断向量表一个表项占用8字节，其中2-3字节是段选择子，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。  （练习6）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。（截图并简要说明）    （练习6）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”。 （截图并简要说明） | | |
| **总结** | 通过本次实验   1. 加强了我们对操作系统启动方式的认识 2. 进一步熟悉了Linux系统的命令行操作 3. 对操作系统中中断等知识有了更深入的了解   4、对于操作系统与硬件的结合也有了一定认识 | | |
| **日期** | 2020.05.31 | **成绩** |  |