|  |  |  |  |
| --- | --- | --- | --- |
| **姓名** | 甘霖 | **学号** | 06 |
| **实验题目** | Lab2实验 | | |
| **实验内容** | 1. **(练习1)了解coure的项目组成**   **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。**  了解操作系统镜像文件ucore.img是如何一步一步生成的  在编译时使用make V=，相当于设置一个标记，使得把make编译执行的过程全部展示出来。  首先，调用了GCC，把一些C的源代码编译成.O的目标文件。    然后，调用了ld，把这些目标文件准换成一个可执行程序，比如下面示例，转换成为了bootloader的一个执行程序bootblock.out。    最后，调用了dd，把bootloader、bootblock和kernel放到虚拟的硬盘uCore.img里面。    在sign.c中，先申请了一个512字节的空间buf，然后将buf初始 化为全0，再将主引导程序写入这个空间，最后，在buf的最后两个字节写入55AA。所以硬盘的主引导扇区的特征是主引导扇区的512个字节的最后两个字节是55AA。    **2、（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。  实验远程调用的方法，启动qemu，并让它进入-S状态。    开启另外一个终端，执行gdb命令，绑定端口1234。    在QEMU窗口中使用 x/10i $pc 查看最近10条指令的反汇编内容。在gdb中使用si命令执行单步，显示位置。可以看到一启动，就处于0xfffffff0的位置，第一条指令是个长跳转指令 。      在初始化位置0x7c00设置实地址断点，测试断点正常   输入 b \*0x7c00设置断点，输入 c 执行到断点。    输入x/10i $pc 查看最近10条指令的反汇编内容，可以看到qemu执行到断点0x7c00。断点工作正常。    再次输入c执行，qemu继续工作。得到结论，断点工作正常。    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    经过比较发现：gdb得到的反汇编代码与bootasm.S和bootblock.asm中的代码基本相一致。    找一个bootloader或内核中的代码位置，设置断点并进行测试  输入 b \*0x7c10设置断点，输入 c 执行到断点。输入x/10i $pc 查看最近10条指令的反汇编内容。    输入stepi，单步执行一条机器指令。再一次输入x/10i $pc 查看最近10条指令的反汇编内容。最后，输入c，qemu继续工作。    **3、（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。**  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   seta20.2:          # 等待8042键盘控制器不忙  inb $0x64, %al  testb $0x2, %al jnz seta20.2  movb $0xdf, %al        # 打开A20   outb %al, $0x60  加载GDTR、初始化GDT表：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  完成进入保护模式，进入bootmain。    call bootmain  **4、（练习4）根据代码，通过阅读bootmain.c，了解bootloader如何加载ELF文件。**  bootloader读取硬盘扇区  readsect函数可以从设备的第secno扇区读取数据到dst位置。 设置读取扇区的数目为、扇区号，读取扇区。   outb(0x1F2, 1);                            outb(0x1F3, secno & 0xFF);   outb(0x1F4, (secno >> 8) & 0xFF);   outb(0x1F5, (secno >> 16) & 0xFF);   outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0)；    outb(0x1F7, 0x20);    读取一个扇区。  insl(0x1F0, dst, SECTSIZE / 4);    readseg函数包装了readsect，使得可以从设备读取任意长度的内容。  bootloader加载ELF格式的OS  读取ELF的头部。  readseg((uintptr\_t)ELFHDR, SECTSIZE \* 8, 0);  判断是否是合法的ELF文件。  if (ELFHDR->e\_magic != ELF\_MAGIC) {            goto bad;}  根据ELFheader和proghdr程序头，读出代码段和数据段，并且加载到相应地方。  ph=(struct proghdr \*)((uintptr\_t)ELFHDR + ELFHDR->e\_phoff);   eph = ph + ELFHDR->e\_phnum;   for (; ph < eph; ph ++) {  readseg(ph->p\_va&0xFFFFFF, ph->p\_memsz, ph->p\_offset);  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))();   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。**   因为ss:ebp指向的堆栈位置储存着跳转之前的ebp，所有，以此为线索可以得到所有使用堆栈的函数ebp。ss:ebp+4指向的是调用时的eip，ss:ebp+8等是参数。  又因为bootloader设置的堆栈从0x7c00开始，使用"call bootmain" 转入bootmain函数，所以，堆栈最深一层值为ebp:0x00007bf8 eip:0x00007d68。      代码分析： 得到当前ebp，eip。  uint32\_t ebp = read\_ebp(), eip = read\_eip();   输出ebp，eip。  cprintf("ebp:0x%08x eip:0x%08x args:", ebp, eip); 设置指针，输出四个参数。  uint32\_t \*args = (uint32\_t \*)ebp + 2;  for (j = 0; j < 4; j ++) {              cprintf("0x%08x ", args[j]);}  调用print\_debuginfo函数完成查找对应函数名并打印至屏幕。   print\_debuginfo(eip - 1); 获取上一层eip，ebp。   eip = ((uint32\_t \*)ebp)[1];    ebp = ((uint32\_t \*)ebp)[0];   1. **(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？**   **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。**  **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”。**  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。  获得IDT表的入口地址。 extern uintptr\_t \_\_vectors[];  在中断门描述符表建立中断门描述符，其中存储了中断处理例程 的代码段GD\_KTEXT和偏移量\_\_vectors[i]，特权级为 DPL\_KERNEL。通过查询idt[i]就可定位到中断服务例程的起始地  址。  for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i ++) {       SETGATE(idt[i],0,GD\_KTEXT,\_\_vectors[i],DPL\_KERNEL); }   通过指令lidt把中断门描述符表的起始地址装入IDTR寄存器中。  lidt(&idt\_pd);  当ticks每加100次后（大约1秒），输出“100 ticks”。 ticks ++;     if (ticks % TICK\_NUM == 0) {      print\_ticks();} | | |
| **总结** | 学会了编译make 文件，学会了使用qumu和gdb调试ucore,学会了从实模式到保护模式的转换方法，了解了中断表。 | | |
| **日期** | 2020.06.01 | **成绩** |  |