|  |  |  |  |
| --- | --- | --- | --- |
| **姓名** | 叶佳璇 | **学号** | 35 |
| **实验题目** | Lab1 | | |
| **实验内容** | **1.（练习1）了解ucore的“项目组成”**  bootloader部分  boot/bootasm.S ：定义并实现了bootloader最先执行的函数start，此函数进行了一定的初始化，完成了从实模式到保护模式的转换，并调用bootmain.c中的bootmain函数。  boot/bootmain.c：定义并实现了bootmain函数实现了通过屏幕、串口和并口显示字符串。bootmain函数加载ucore操作系统到内存，然后跳转到ucore的入口处执行。  boot/asm.h：是bootasm.S汇编文件所需要的头文件，主要是一些与X86保护模式的段访问方式相关的宏定义。  ucore操作系统部分  系统初始化部分：  kern/init/init.c：ucore操作系统的初始化启动代码  内存管理部分：  kern/mm/memlayout.h：ucore操作系统有关段管理（段描述符编号、段号等）的一些宏定义  kern/mm/mmu.h：ucore操作系统有关X86 MMU等硬件相关的定义，包括EFLAGS寄存器中各位的含义，应用/系统段类型，中断门描述符定义，段描述符定义，任务状态段定义，NULL段声明的宏SEG\_NULL, 特定段声明的宏SEG，设置中 断门描述符的宏SETGATE（在练习6中会用到）  kern/mm/pmm.[ch]：设定了ucore操作系统在段机制中要用到的全局变量：任务状态段ts，全局描述符表 gdt[]，加载全局描述符表寄存器的函数lgdt，临时的内核栈stack0；以及对全局描述符表和任务状态段的初始化函数gdt\_init  外设驱动部分：  kern/driver/intr.[ch]：实现了通过设置CPU的eflags来屏蔽和使能中断的函数；  kern/driver/picirq.[ch]：实现了对中断控制器8259A的初始化和使能操作；  kern/driver/clock.[ch]：实现了对时钟控制器8253的初始化操作；- kern/driver/console.[ch]：实现了对串口和键盘的中断方式的处理操作；  中断处理部分：  kern/trap/vectors.S：包括256个中断服务例程的入口地址和第一步初步处理实现。注意，此文件是由tools/vector.c在编译ucore期间动态生成的；  kern/trap/trapentry.S：紧接着第一步初步处理后，进一步完成第二步初步处理；并且有恢复中断上下文的处理，即中断处理完毕后的返回准备工作；  kern/trap/trap.[ch]：紧接着第二步初步处理后，继续完成具体的各种中断处理操作；  内核调试部分：  kern/debug/kdebug.[ch]：提供源码和二进制对应关系的查询功能，用于显示调用栈关系。其中补全print\_stackframe函数是需要完成的练习。其他实现部分不必深究。  kern/debug/kmonitor.[ch]：实现提供动态分析命令的kernel monitor，便于在ucore出现bug或问题后，能够进入kernel monitor中，查看当前调用关系。实现部分不必深究。  kern/debug/panic.c | assert.h：提供了panic函数和assert宏，便于在发现错误后，调用kernel monitor。大家可在编程实验中充分利用assert宏和panic函数，提高查找错误的效率。  公共库部分  libs/defs.h：包含一些无符号整型的缩写定义。  Libs/x86.h：一些用GNU C嵌入式汇编实现的C函数（由于使用了inline关键字，所以可以理解为宏）。  工具部分  Makefile和function.mk：指导make完成整个软件项目的编译，清除等工作。  sign.c：一个C语言小程序，是辅助工具，用于生成一个符合规范的硬盘主引导扇区。  tools/vector.c：生成vectors.S，此文件包含了中断向量处理的统一实现。  **2.（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。**        cc kern/init/init.c //编译init.c  cc kern/libs/readline.c //编译readline.c  cc kern/libs/stdio.c //编译stdlio.c  cc kern/debug/kdebug.c //编译kdebug.c      cc kern/debug/kmonitor.c //编译komnitor.c  cc kern/debug/panic.c //编译panic.c  cc kern/driver/clock.c //编译clock.c  cc kern/driver/console.c //编译console.c  cc kern/driver/intr.c //编译intr.c  cc kern/driver/picirq.c //编译prcirq.c  cc kern/trap/trap.c //编译trap.c  cc kern/trap/trapentry.S //编译trapentry.  cc kern/trap/vectors.S //编译vectors.S  cc kern/mm/pmm.c //编译pmm.c  cc libs/printfmt.c //编译printfmt.c    cc libs/string.c //编译string.c  ld bin/kernel //链接成kernel  cc boot/bootasm.S //编译bootasm.c  cc boot/bootmain.c //编译bootmain.c  cc tools/sign.c //编译sign.c  ld bin/bootblock //根据sign规范生成bootblock  编译过程  1 编译所有生成bin/kernel所需的文件  2 链接生成bin/kernel  3 编译bootasm.S bootmain.c sign.c  4 根据sign规范生成obj/bootblock.o  5 生成ucore.img    Sign.c:修饰编译生成的bootloader文件  **3.（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）**  修改gdbinit内容    Lab1执行    单步追踪    修改gdbnit        查看bootasm.S文件    查看bootblock.asm    结果一样  修改gdbnit设置断点      **4.（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）**  关中断和清除数据段寄存器  cli //关中断  cld //清除方向标志  xorw %ax, %ax //ax清0  movw %ax, %ds //ds清0  movw %ax, %es //es清0  movw %ax, %ss //ss清0  初始时A20为0，访问超过1MB的地址时，就会从0循环计数，将A20地址线置为1之后，才可以访问4G内存。A20地址位由8042控制，8042有2个有两个I/O端口：0x60和0x64。  打开流程：  等待8042 Input buffer为空；  发送Write 8042 Output Port （P2）命令到8042 Input buffer；  等待8042 Input buffer为空；  将8042 Output Port（P2）得到字节的第2位置1，然后写入8042 Input buffer；    seta20.1: //等待8042键盘控制器不忙  inb $0x64, %al //从0x64端口中读入一个字节到al中  testb $0x2, %al //测试al的第2位  jnz seta20.1 //al的第2位为0，则跳出循环  movb $0xd1, %al //将0xd1写入al中  outb %al, $0x64 //将0xd1写入到0x64端口中  seta20.2: //等待8042键盘控制器不忙  inb $0x64, %al //从0x64端口中读入一个字节到al中  testb $0x2, %al //测试al的第2位  jnz seta20.2 //al的第2位为0，则跳出循环  movb $0xdf, %al //将0xdf入al中  outb %al, $0x60 //将0xdf入到0x64端口中，打开A20    lgdt gdtdesc //载入GDT表  movl %cr0, %eax //加载cro到eax  orl $CR0\_PE\_ON, %eax //将eax的第0位置为1  movl %eax, %cr0 //将cr0的第0位置为1  通过长跳转更新cs的基地址    设置段寄存器，并建立堆栈  转到保护模式完成，进入boot主方法  **5.（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）**  Boot loader读取硬盘扇区  1、等待磁盘准备好；  2、发出读取扇区的命令；  3、等待磁盘准备好；  4、把磁盘扇区数据读到指定内存。    outb(0x1F2, 1); //读取一个扇区  outb(0x1F3, secno & 0xFF); //要读取的扇区编号  outb(0x1F4, (secno >> 8)&0xFF);//用来存放读写柱面的低8位字节  outb(0x1F5, (secno >> 16)&0xFF);//用来存放读写柱面的高2位字节    加载elf格式的os    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))();  总结一下就是：  1从硬盘读了8个扇区数据到内存0x10000处，并把这里强制转换成elfhdr使用；  2校验e\_magic字段；  3根据偏移量分别把程序段的数据读取到内存中。  **（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。）**      执行make qemu    **7.（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）**    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字节拼成偏移量。  通过段选择子去GDT中找到相应的基地址，然后基地址加上偏移量就是中断处理程序地址  **8.（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）**      **9.（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）**    **加入string.h**    **定义变量**    **Trap\_dispatch 实现**    **10.参考答案labcodes\_answer/lab1,并在labcodes/lab1中完成challenge1内容，并简要说明实现的过程（设计到哪些函数，分别在哪个文件中，执行中断的过程中，先后使用了哪些函数。）。** 扩展练习 Challenge 1（需要编程） 扩展proj4,增加syscall功能，即增加一用户态函数（可执行一特定系统调用：获得时钟计数值），当内核初始完毕后，可从内核态返回到用户态的函数，而用户态的函数又通过系统调用得到内核态的服务（通过网络查询所需信息，可找老师咨询。如果完成，且有兴趣做代替考试的实验，可找老师商量）。需写出详细的设计和分析报告。完成出色的可获得适当加分。  提示： 规范一下 challenge 的流程。  kern\_init 调用 switch\_test，该函数如下：  +  static void  switch\_test(void) {  print\_cur\_status(); // print 当前 cs/ss/ds 等寄存器状态  cprintf("+++ switch to user mode +++\n");  switch\_to\_user(); // switch to user mode  print\_cur\_status();  cprintf("+++ switch to kernel mode +++\n");  switch\_to\_kernel(); // switch to kernel mode  print\_cur\_status();  }  switchto\* 函数建议通过 中断处理的方式实现。主要要完成的代码是在 trap 里面处理 T\_SWITCH\_TO\* 中断，并设置好返回的状态。  在 lab1 里面完成代码以后，执行 make grade 应该能够评测结果是否正确。 | | |
| **总结** | （通过练习掌握到的知识，按条目列出。） | | |
| **日期** | 2020.05.XX | **成绩** |  |