|  |  |  |  |
| --- | --- | --- | --- |
| **姓名** | 黄庆华 | **学号** | 11 |
| **实验题目** | 第二次实验  Lab1 | | |
| **实验内容** | **实验过程：**  **1.（练习1）了解ucore的“项目组成”。**  计算机原理  CPU的编址与寻址: 基于分段机制的内存管理  CPU的中断机制  外设：串口/并口/CGA，时钟，硬盘  Bootloader软件  编译运行bootloader的过程  调试bootloader的方法  PC启动bootloader的过程  ELF执行文件的格式和加载  外设访问：读硬盘，在CGA上显示字符串  ucore OS软件  编译运行ucore OS的过程  ucore OS的启动过程  调试ucore OS的方法  函数调用关系：在汇编级了解函数调用栈的结构和处理过程  中断管理：与软件相关的中断处理  外设管理：时钟  **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。**  1、生成ucore.img需要kernel和bootblock  生成ucore.img的代码如下：  $(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  $(call create\_target,ucore.img)  首先先创建一个大小为10000字节的块儿，然后再将bootblock拷贝过去。  生成ucore.img需要先生成kernel和bootblock  2、生成kernel  而生成kernel的代码如下：  $(kernel): tools/kernel.ld  $(kernel): $(KOBJS)  @echo "bbbbbbbbbbbbbbbbbbbbbb$(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)    通过make V=指令得到执行的具体命令如下：  ld -m elf\_i386 -nostdlib -T tools/kernel.ld -o bin/kernel obj/kern/init/init.o obj/kern/libs/readline.o obj/kern/libs/stdio.o obj/kern/debug/kdebug.o obj/kern/debug/kmonitor.o obj/kern/debug/panic.o obj/kern/driver/clock.o obj/kern/driver/console.o obj/kern/driver/intr.o obj/kern/driver/picirq.o obj/kern/trap/trap.o obj/kern/trap/trapentry.o obj/kern/trap/vectors.o obj/kern/mm/pmm.o obj/libs/printfmt.o obj/libs/string.o  1  然后根据其中可以看到，要生成kernel，需要用GCC编译器将kern目录下所有的.c文件全部编译生成的.o文件的支持。具体如下：  obj/kern/init/init.o  obj/kern/libs/readline.o  obj/kern/libs/stdio.o  obj/kern/debug/kdebug.o  obj/kern/debug/kmonitor.o  obj/kern/debug/panic.o  obj/kern/driver/clock.o  obj/kern/driver/console.o  obj/kern/driver/intr.o  obj/kern/driver/picirq.o  obj/kern/trap/trap.o  obj/kern/trap/trapentry.o  obj/kern/trap/vectors.o  obj/kern/mm/pmm.o  obj/libs/printfmt.o  obj/libs/string.o  3、生成bootblock  而生成bootblock的代码如下：  $(bootblock): $(call toobj,$(bootfiles)) | $(call totarget,sign)  @echo "========================$(call toobj,$(bootfiles))"  @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)  同样根据make V=指令打印的结果，得到要生成bootblock，首先需要生成bootasm.o、bootmain.o、sign，  下列代码为生成bootasm.o、bootmain.o的代码，由宏定义批量实现了。  bootfiles = $(call listf\_cc,boot)  $(foreach f,$(bootfiles),$(call cc\_compile,$(f),$(CC),$(CFLAGS) -Os -nostdinc))    而实际的命令在make V=指令结果里可以看到。  下述是由bootasm.S生成bootasm.o的具体命令：  gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootasm.S -o obj/boot/bootasm.o  1  下述是由bootmain.c生成bootmain.o的具体命令  gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootmain.c -o obj/boot/bootmain.o  至于上述命令的具体参数，查阅资料罗列如下：  - -ggdb 生成可供gdb使用的调试信息  - -m32 生成适用于32位环境的代码  - -gstabs 生成stabs格式的调试信息  - -nostdinc 不使用标准库  - -fno-stack-protector 不生成用于检测缓冲区溢出的代码  - -Os 为减小代码大小而进行优化  - -I  添加搜索头文件的路径  - -fno-builtin 不进行builtin函数的优化  下列代码为生成sign的代码  $(call add\_files\_host,tools/sign.c,sign,sign)  $(call create\_target\_host,sign,sign)  下面是生成sign具体的命令:  gcc -Itools/ -g -Wall -O2 -c tools/sign.c -o obj/sign/tools/sign.o  gcc -g -Wall -O2 obj/sign/tools/sign.o -o bin/sign  有了上述的bootasm.o、bootmain.o、sign。  接下来就可以生成bootblock了，实际命令如下：  ld -m elf\_i386 -nostdlib -N -e start -Ttext 0x7C00 obj/boot/bootasm.o obj/boot/bootmain.o -o obj/bootblock.o  参数解释如下：（不重复解释）  - -m 模拟为i386上的连接器  - -N 设置代码段和数据段均可读写  - -e 指定入口  - -Ttext 制定代码段开始位置  **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）**  首先通过make qemu指令运行出等待调试的qemu虚拟机，然后再打开一个终端，通过下述命令连接到qemu虚拟机：  gdb  target remote 127.0.0.1:1234    进入到调试界面：  输入si命令单步调试，  这是另一个终端会打印下一条命令的地址和内容：  然后输入b\*0x7c00在初始化位置地址0x7c00设置上断点，如下：  然后输入continue使之继续运行：  这时成功在0x7c00处停止运行，然后我们查看此处的反汇编代码，如下：  对比此时bootasm.S中的起始代码，发现确实是一样的  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） **1、关闭中断，将各个段寄存器重置它先将各个寄存器置0**  cli # Disable interrupts  cld # String operations increment  xorw %ax, %ax # Segment number zero  movw %ax, %ds # -> Data Segment  movw %ax, %es # -> Extra Segment  movw %ax, %ss # -> Stack Segment  **2、开启A20**  然后就是将A20置1，这里简单解释一下A20，当 A20 地址线控制禁止时，则程序就像在 8086 中运行，1MB 以上的地是不可访问的。而在保护模式下 A20 地址线控制是要打开的，所以需要通过将键盘控制器上的A20线置于高电位，使得全部32条地址线可用。  seta20.1:  inb $0x64, %al # 读取状态寄存器,等待8042键盘控制器闲置  testb $0x2, %al # 判断输入缓存是否为空  jnz seta20.1  movb $0xd1, %al # 0xd1表示写输出端口命令，参数随后通过0x60端口写入  outb %al, $0x64  seta20.2:  inb $0x64, %al  testb $0x2, %al  jnz seta20.2  movb $0xdf, %al # 通过0x60写入数据11011111 即将A20置1  outb %al, $0x60  **3、加载GDT表** lgdt gdtdesc  **4、将CR0的第0位置1**  movl %cr0, %eax  orl $CR0\_PE\_ON, %eax  movl %eax, %cr0  **5、长跳转到32位代码段，重装CS和EIP**  ljmp $PROT\_MODE\_CSEG, $protcseg  **6、重装DS、ES等段寄存器等**  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  7、转到保护模式完成，进入boot主方法  movl $0x0, %ebp  movl $start, %esp  call bootmain  **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）**  bootmain(void) {  readseg((uintptr\_t)ELFHDR, SECTSIZE \* 8, 0);  if (ELFHDR->e\_magic != ELF\_MAGIC) {  goto bad;  }  struct proghdr \*ph, \*eph;  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);  }  ((void (\*)(void))(ELFHDR->e\_entry & 0xFFFFFF))();  bad:  outw(0x8A00, 0x8A00);  outw(0x8A00, 0x8E00);  while (1);  }  bootloader读取硬盘扇区  根据上述bootmain函数分析，首先是由readseg函数读取硬盘扇区，而readseg函数则循环调用了真正读取硬盘扇区的函数readsect来每次读出一个扇区 ，如下，详细的解释看代码中的注释：  readsect(void \*dst, uint32\_t secno) {  waitdisk(); // 等待硬盘就绪  // 写地址0x1f2~0x1f5,0x1f7,发出读取磁盘的命令  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);  waitdisk();  insl(0x1F0, dst, SECTSIZE / 4);//读取一个扇区  }  bootloader加载 ELF格式的 OS  读取完磁盘之后，开始加载ELF格式的文件。详细的解释看代码中的注释。  bootmain(void) {  ..........  //首先判断是不是ELF  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，这里读取出来将之存入ph  ph = (struct proghdr \*)((uintptr\_t)ELFHDR + ELFHDR->e\_phoff);  eph = ph + ELFHDR->e\_phnum;  //按照程序头表的描述，将ELF文件中的数据载入内存  for (; ph < eph; ph ++) {  readseg(ph->p\_va & 0xFFFFFF, ph->p\_memsz, ph->p\_offset);  }  //根据ELF头表中的入口信息，找到内核的入口并开始运行  ((void (\*)(void))(ELFHDR->e\_entry & 0xFFFFFF))();  bad:  ..........  }  **6.（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。）**  1、函数堆栈的原理  理解函数堆栈最重要的两点是：栈的结构，以及EBP寄存器的作用。  一个函数调用动作可分解为零到多个 PUSH指令（用于参数入栈）和一个 CALL 指令。CALL 指令内部其实还暗含了一个将返回地址压栈的动作，这是由硬件完成的。几乎所有本地编译器都会在每个函数体之前插入类似如下的汇编指令：  pushl %ebp  movl %esp,%ebp  这两条汇编指令的含义是：首先将ebp 寄存器入栈，然后将栈顶指针 esp 赋值给 ebp。  movl %esp %ebp这条指令表面上看是用esp覆盖 ebp原来的值，其实不然。因为给 ebp赋值之前，  原ebp 值已经被压栈（位于栈顶），而新的ebp又恰恰指向栈顶。此时ebp寄存器就已经处于一个  非常重要的地位，该寄存器中存储着栈中的一个地址（原 ebp入栈后的栈顶），从该地址为基准，  向上（栈底方向）能获取返回地址、参数值，向下（栈顶方向）能获取函数局部变量值，而该地址  处又存储着上一层函数调用时的ebp值。  现在做一下更完整的解释：  函数调用大概包括以下几个步骤：  - 1、参数入栈：将参数从右向左（或从右向左）依次压入系统栈中。  - 2、返回地址入栈：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行。  - 3、代码区跳转：处理器从当前代码区跳转到被调用函数的入口处。  - 4、栈帧调整  - 4.1保存当前栈帧状态值，已备后面恢复本栈帧时使用（EBP入栈）。  - 4.2将当前栈帧切换到新栈帧（将ESP值装入EBP，更新栈帧底部）。  - 4.3给新栈帧分配空间（把ESP减去所需空间的大小，抬高栈顶）。  而函数返回大概包括以下几个步骤：  - 1、保存返回值，通常将函数的返回值保存在寄存器EAX中。  - 2、弹出当前帧，恢复上一个栈帧。  - 2.1在堆栈平衡的基础上，给ESP加上栈帧的大小，降低栈顶，回收当前栈帧的空间  - 2.2将当前栈帧底部保存的前栈帧EBP值弹入EBP寄存器，恢复出上一个栈帧。  - 2.3将函数返回地址弹给EIP寄存器。  - 3、跳转：按照函数返回地址跳回母函数中继续执行。  而由此我们可以直接根据ebp就能读取到各个栈帧的地址和值，一般而言，ss:[ebp+4]处为返回地址，ss:[ebp+8]处为第一个参数值（最后一个入栈的参数值，此处假设其占用 4 字节内存，对应32位系统），ss:[ebp-4]处为第一个局部变量，ss:[ebp]处为上一层 ebp 值。  2、print\_stackframe函数的实现  首先我们直接看到print\_stackframe函数的注释：  void print\_stackframe(void) {  /\* LAB1 YOUR CODE : STEP 1 \*/  /\* (1) call read\_ebp() to get the value of ebp. the type is (uint32\_t);  \* (2) call read\_eip() to get the value of eip. the type is (uint32\_t);  \* (3) from 0 .. STACKFRAME\_DEPTH  \* (3.1) printf value of ebp, eip  \* (3.2) (uint32\_t)calling arguments [0..4] = the contents in address (unit32\_t)ebp +2 [0..4]  \* (3.3) cprintf("\n");  \* (3.4) call print\_debuginfo(eip-1) to print the C calling function name and line number, etc.  \* (3.5) popup a calling stackframe  \* NOTICE: the calling funciton's return addr eip = ss:[ebp+4]  \* the calling funciton's ebp = ss:[ebp]  \*/  }  这样我们直接根据注释以及之前的相关知识就能比较简单的编写成程序，如下所示：  void print\_stackframe(void) {  uint32\_t ebp=read\_ebp();//(1) call read\_ebp() to get the value of ebp. the type is (uint32\_t)  uint32\_t eip=read\_eip();//(2) call read\_eip() to get the value of eip. the type is (uint32\_t)  int i;  for(i=0;i<STACKFRAME\_DEPTH&&ebp!=0;i++){//(3) from 0 .. STACKFRAME\_DEPTH  cprintf("ebp:0x%08x eip:0x%08x ",ebp,eip);//(3.1)printf value of ebp, eip  uint32\_t \*tmp=(uint32\_t \*)ebp+2;  cprintf("arg :0x%08x 0x%08x 0x%08x 0x%08x",\*(tmp+0),\*(tmp+1),\*(tmp+2),\*(tmp+3));//(3.2)(uint32\_t)calling arguments [0..4] = the contents in address (unit32\_t)ebp +2 [0..4]  cprintf("\n");//(3.3) cprintf("\n");  print\_debuginfo(eip-1);//(3.4) call print\_debuginfo(eip-1) to print the C calling function name and line number, etc.  eip=((uint32\_t \*)ebp)[1];  ebp=((uint32\_t \*)ebp)[0];//(3.5) popup a calling stackframe  }  }  **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）**  **中断描述符表一个表项占8字节。其中0~15位和48~63位分别为offset的低16位和高16位。16~31位为段选择子。通过段选择子获得段基址，加上段内偏移量即可得到中断处理代码的入口。**  这里这里主要就是实现对中断向量表的初始化。  注释如下：  void idt\_init(void) {  /\* LAB1 YOUR CODE : STEP 2 \*/  /\* (1) Where are the entry addrs of each Interrupt Service Routine (ISR)?  \* All ISR's entry addrs are stored in \_\_vectors. where is uintptr\_t \_\_vectors[] ?  \* \_\_vectors[] is in kern/trap/vector.S which is produced by tools/vector.c  \* (try "make" command in lab1, then you will find vector.S in kern/trap DIR)  \* You can use "extern uintptr\_t \_\_vectors[];" to define this extern variable which will be used later.  \* (2) Now you should setup the entries of ISR in Interrupt Description Table (IDT).  \* Can you see idt[256] in this file? Yes, it's IDT! you can use SETGATE macro to setup each item of IDT  \* (3) After setup the contents of IDT, you will let CPU know where is the IDT by using 'lidt' instruction.  \* You don't know the meaning of this instruction? just google it! and check the libs/x86.h to know more.  \* Notice: the argument of lidt is idt\_pd. try to find it!  \*/  重点就是两步  第一步，声明\_\_vertors[],其中存放着中断服务程序的入口地址。这个数组生成于vertor.S中。  第二步，填充中断描述符表IDT。  第三部，加载中断描述符表。  对应到代码中如下所示：  void idt\_init(void) {  extern uintptr\_t \_\_vectors[];//声明\_\_vertors[]  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);  lidt(&idt\_pd);//使用lidt指令加载中断描述符表  }  这里的SETGATE在mmu.h中有定义，  #define SETGATE(gate, istrap, sel, off, dpl)  简单解释一下参数  gate：为相应的idt[]数组内容，处理函数的入口地址  istrap：系统段设置为1，中断门设置为0  sel：段选择子  off：为\_\_vectors[]数组内容  dpl：设置特权级。这里中断都设置为内核级，即第0级  这里根据指导书查看函数trap\_dispatch，发现print\_ticks()子程序已经被实现了，所以我们直接进行判断输出即可，如下（见注释）：  ........  ........  case IRQ\_OFFSET + IRQ\_TIMER:  ticks ++; //每一次时钟信号会使变量ticks加1  if (ticks==TICK\_NUM) {//TICK\_NUM已经被预定义成了100，每到100便调用print\_ticks()函数打印  ticks-=TICK\_NUM;  print\_ticks();  }  break;  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）  我的源代码  extern uintptr\_t \_\_vectors[];  int i;  //初始化idt  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);  SETGATE(idt[T\_SWITCH\_TOU],0,GD\_KTEXT,\_\_vectors[T\_SWITCH\_TOU],DPL\_KERNEL);  lidt(&idt\_pd);    代码很简单，首先引入中断处理函数的入口地址\_\_vectors[],这个变量在vector.s里面生成的  ，然后初始化idt中断描述符表，最后根据提示用lidt函数告知cpu IDT表的位置。  1.写完后，发现这中断向量表的代码贼短，其实里面涉及到的东西不少哟。  我们看，第一句，声明一个vectors数组，这个vector在vector.S里面定义的，意思是通过这个指针可以跳转到该中断处理的地点。  .globl vector20  vector20:  pushl $0  pushl $20  jmp \_\_alltraps  举个例子，上面的代码是中断号为20的跳转代码，首先会把该中断信息（中断号）进行压栈，在进入到统一的中断预处理函数。接下来跳转到\_\_alltraps，该函数的作用很简单：保存好用户空间的上下文（也就是一些寄存器变量）。并切换到内核的上下文。  2.setgate这个函数的作用是设置正确的interrupt/trap gate 描述符。接下来，我来介绍一下该函数的参数。  //gate: Gate descriptors for interrupts and traps  //istrap : 0 is interrupts ,else is traps  //sel : the off's segment ，usual kernel text,the value is GD\_KTEXT  //off : offset in segment ,函数入口地址。  //dpl ： 特权级。  #define SETGATE(gate, istrap, sel, off, dpl) { \  (gate).gd\_off\_15\_0 = (uint32\_t)(off) & 0xffff; \  (gate).gd\_ss = (sel); \  (gate).gd\_args = 0; \  (gate).gd\_rsv1 = 0; \  (gate).gd\_type = (istrap) ? STS\_TG32 : STS\_IG32; \  (gate).gd\_s = 0; \  (gate).gd\_dpl = (dpl); \  (gate).gd\_p = 1; \  (gate).gd\_off\_31\_16 = (uint32\_t)(off) >> 16; \  }  /\* Gate descriptors for interrupts and traps \*/  struct gatedesc {  unsigned gd\_off\_15\_0 : 16; // low 16 bits of offset in segment  unsigned gd\_ss : 16; // segment selector  unsigned gd\_args : 5; // # args, 0 for interrupt/trap gates  unsigned gd\_rsv1 : 3; // reserved(should be zero I guess)  unsigned gd\_type : 4; // type(STS\_{TG,IG32,TG32})  unsigned gd\_s : 1; // must be 0 (system)  unsigned gd\_dpl : 2; // descriptor(meaning new) privilege level  unsigned gd\_p : 1; // Present  unsigned gd\_off\_31\_16 : 16; // high bits of offset in segment  };  3.注意需要对T\_SWITCH\_TOK的发生时机是在用户空间的，所以对应的dpl需要修改为DPL\_USER。  4.lidt将idt的首地址和size装进idtr寄存器。  **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）**  SETGATE函数的实现：  #define SETGATE(gate, istrap, sel, off, dpl) { \  (gate).gd\_off\_15\_0 = (uint32\_t)(off) & 0xffff; \  (gate).gd\_ss = (sel); \  (gate).gd\_args = 0; \  (gate).gd\_rsv1 = 0; \  (gate).gd\_type = (istrap) ? STS\_TG32 : STS\_IG32; \  (gate).gd\_s = 0; \  (gate).gd\_dpl = (dpl); \  (gate).gd\_p = 1; \  (gate).gd\_off\_31\_16 = (uint32\_t)(off) >> 16; \  }  宏定义和数组说明：  #define GD\_KTEXT ((SEG\_KTEXT) << 3) // kernel text  #define DPL\_KERNEL (0)  #define DPL\_USER (3)  #define T\_SWITCH\_TOK 121 // user/kernel switch  static struct gatedesc idt[256] = {{0}};  idt\_init函数的实现：  void  idt\_init(void) {  extern uintptr\_t \_\_vectors[]; //保存在vectors.S中的256个中断处理例程的入口地址数组  int i;  //使用SETGATE宏，对中断描述符表中的每一个表项进行设置  for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i ++) { //IDT表项的个数  //在中断门描述符表中通过建立中断门描述符，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。这样通过查询idt[i]就可定位到中断服务例程的起始地址。  SETGATE(idt[i], 0, GD\_KTEXT, \_\_vectors[i], DPL\_KERNEL);  }  SETGATE(idt[T\_SWITCH\_TOK], 0, GD\_KTEXT,  \_\_vectors[T\_SWITCH\_TOK], DPL\_USER);  //建立好中断门描述符表后，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，从而完成中段描述符表的初始化工作。  lidt(&idt\_pd);  首先加入 string.h头文件，为了使用memmove函数  void \*memmove(void \*dst, const void \*src, size\_t n);  1  定义变量：  struct trapframe switchk2u, \*switchu2k;  1  结构体 trapframe  struct trapframe {  struct pushregs tf\_regs;  uint16\_t tf\_gs;  uint16\_t tf\_padding0;  uint16\_t tf\_fs;  uint16\_t tf\_padding1;  uint16\_t tf\_es;  uint16\_t tf\_padding2;  uint16\_t tf\_ds;  uint16\_t tf\_padding3;  uint32\_t tf\_trapno;  /\* below here defined by x86 hardware \*/  uint32\_t tf\_err;  uintptr\_t tf\_eip;  uint16\_t tf\_cs;  uint16\_t tf\_padding4;  uint32\_t tf\_eflags;  /\* below here only when crossing rings, such as from user to kernel \*/  uintptr\_t tf\_esp;  uint16\_t tf\_ss;  uint16\_t tf\_padding5;  } \_\_attribute\_\_((packed));  宏定义：  #define IRQ\_OFFSET 32  #define IRQ\_TIMER 0  #define IRQ\_KBD 1  #define IRQ\_COM1 4  #define T\_SWITCH\_TOU 120  #define USER\_CS ((GD\_UTEXT) | DPL\_USER)  #define USER\_DS ((GD\_UDATA) | DPL\_USER)  #define KERNEL\_DS ((GD\_KDATA) | DPL\_KERNEL)  #define TICK\_NUM 100  print\_ticks函数  static void print\_ticks() {  cprintf("%d ticks\n",TICK\_NUM);  #ifdef DEBUG\_GRADE  cprintf("End of Test.\n");  panic("EOT: kernel seems ok.");  #endif  }  trap\_dispatch函数的实现：  static void  trap\_dispatch(struct trapframe \*tf) {  char c;  switch (tf->tf\_trapno) {  case IRQ\_OFFSET + IRQ\_TIMER:  ticks ++;  if (ticks % TICK\_NUM == 0) {  print\_ticks();  }  break;  //下面的代码不用我们实现  case IRQ\_OFFSET + IRQ\_COM1:  c = cons\_getc();  cprintf("serial [%03d] %c\n", c, c);  break;  case IRQ\_OFFSET + IRQ\_KBD:  c = cons\_getc();  cprintf("kbd [%03d] %c\n", c, c);  break;  case T\_SWITCH\_TOU:  if (tf->tf\_cs != USER\_CS) {  switchk2u = \*tf;  switchk2u.tf\_cs = USER\_CS;  switchk2u.tf\_ds = switchk2u.tf\_es = switchk2u.tf\_ss = USER\_DS;  switchk2u.tf\_esp = (uint32\_t)tf + sizeof(struct trapframe) - 8;  switchk2u.tf\_eflags |= FL\_IOPL\_MASK;  \*((uint32\_t \*)tf - 1) = (uint32\_t)&switchk2u;  }  break;  case T\_SWITCH\_TOK:  if (tf->tf\_cs != KERNEL\_CS) {  tf->tf\_cs = KERNEL\_CS;  tf->tf\_ds = tf->tf\_es = KERNEL\_DS;  tf->tf\_eflags &= ~FL\_IOPL\_MASK;  switchu2k = (struct trapframe \*)(tf->tf\_esp - (sizeof(struct trapframe) - 8));  memmove(switchu2k, tf, sizeof(struct trapframe) - 8);  \*((uint32\_t \*)tf - 1) = (uint32\_t)switchu2k;  }  break;  case IRQ\_OFFSET + IRQ\_IDE1:  case IRQ\_OFFSET + IRQ\_IDE2:  break;  default:  if ((tf->tf\_cs & 3) == 0) {  print\_trapframe(tf);  panic("unexpected trap in kernel.\n");  }  }  }  **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** | 对比两个lab的tools/kernel.ld文件，发觉lab2的链接脚本汇总在定义edata前，先用ALIGN命令将位置设置在能被0x1000整除的位置，这样恰好将.got.plt, data.rel.ro.local这些段跳过了，因此(edata, end)这段内存恰好只包含.bss段，这时memset就没问题了！定位问题的一种思路：如果有两份代码，一份有问题，另一份正常，那么可以使用对比法。定位问题的常规方法：收集信息、理解原理、推测原因（将所有可能的原因都列举出来，逐一排查）。如果不是之前做mit 6.828 lab遇到过同样的Bug，估计定位起来更艰难，因为我差别就放弃怀疑memset语句有问题了。这个问题的棘手之处也在于：memset时没立即出错，等到后面初始化GDT时才出错。总之，经验很重要，以及不能随便放过任何一个可能。  可以推测lab2的写法是比lab1省内存的，因为使用宏代替了全局变量。但这点差异足够大到lab1链接不通过吗？先将lab2的写法同步到lab1，再make一把，发现果然可以了，boot/bootblock.out的size由600字节减少到488字节，少了112字节。 | | |
| **日期** | 2020.05.31 | **成绩** |  |