《操作系统原理》lab1实验报告

专业：管理学院会计学（辅修）

姓名：莫书琪

学号：17318086

1. 实验目的

操作系统是一个软件，也需要通过某种机制加载并运行它。在这里我们将通过另外一个更加简单的软件-bootloader 来完成这些工作。为此，我们需要完成一个能够切换到x86 的保护模式并显示字符的bootloader，为启动操作系统ucore 做准备。lab1 提供了一个非常小的bootloader 和ucoreOS，整个bootloader 执行代码小于512 个字节，这样才能放到硬盘的主引导扇区中。通过分析和实现这个bootloader 和ucore OS，可以了解到：

- 基于分段机制的存储管理
- 设备管理的基本概念
- PC 启动bootloader 的过程
- bootloader 的文件组成
- 编译运行 bootloader 的过程
- 调试 bootloader 的方法
- ucore OS 的启动过程
- 在汇编级了解栈的结构和处理过程
- 中断处理机制
- 通过串口/并口/CGA 输出字符的方法

2. 实验过程

练习1：理解通过make生成执行文件的过程

实验过程：在终端输入make V=命令

生成ucore.img的核心代码

根据Makefile文件中创建ucore.img的代码，ucore.img由三部分组成：/dev/zero（一个character special file），bootblock和kernel。

```
# create ucore.img
UCOREIMG    := $(call totarget,ucore.img)

$(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)
```

1. 生成kernel

- 对kernel的各个依赖项分别进行编译：以生成init.o为例

- 关键参数：-fno-builtin使与c语言内建函数同名的自定义函数正常编译，-Wall生成所有警告信息，-ggdb可以尽可能生成gdb可以使用的调试信息，-gstabs 此选项以stabs格式声称调试信息，-m32生成适用于32位环境的代码，-nostdinc 使编译器不在系统缺省的头文件目录里面找头文件，-fno-stack-protector不生成用于检测缓冲区溢出的代码，-I<dir> 添加搜索头文件的路径

```
gcc -Ikern/init/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-pr
otector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/in
it/init.c -o obj/kern/init/init.o
```

- 将编译生成的.o文件，利用链接指令ld，生成操作系统内核可执行文件kernel
  - 关键参数：-m模拟指定的连接器，-nostdlib不适用标准库，-T指定命令文件，-o指定输出文件的名称，init.o等为生成kernel所需要的文件

```
+ ld bin/kernel
ld -m    elf_i386 -nostdlib -T tools/kernel.ld -o bin/kernel  obj/kern/init/ini
t.o obj/kern/libs/stdio.o obj/kern/libs/readline.o obj/kern/debug/panic.o obj/k
ern/debug/kdebug.o obj/kern/debug/kmonitor.o obj/kern/driver/clock.o obj/kern/d
river/console.o obj/kern/driver/picirq.o obj/kern/driver/intr.o obj/kern/trap/t
rap.o obj/kern/trap/vectors.o obj/kern/trap/trapentry.o obj/kern/mm/pmm.o  obj/
libs/string.o obj/libs/printfmt.o
```

2. 生成bootblock

- 编译依赖项：bootasm.o和bootmain.o

```
+ cc boot/bootasm.S
gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-protect
or -Ilibs/ -Os -nostdinc -c boot/bootasm.S -o obj/boot/bootasm.o
+ cc boot/bootmain.c
gcc -Iboot/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-protect
or -Ilibs/ -Os -nostdinc -c boot/bootmain.c -o obj/boot/bootmain.o
```

- 将bootasm.o和bootmain.o链接成可执行文件，bootloader的起始地址为0x700，大小为512字节，正好为一个磁盘扇区的大小
  - 关键参数：-N设置代码段和数据段均可读写
  - sign.c将488字节的bootblock.out读到了512字节的bin/bootblock

```
+ ld bin/bootblock
ld -m    elf_i386 -nostdlib -N -e start -Ttext 0x7C00 obj/boot/bootasm.o obj/bo
ot/bootmain.o -o obj/bootblock.o
'obj/bootblock.out' size: 488 bytes
build 512 bytes boot sector: 'bin/bootblock' success!
```

一个被系统认为是符合规范的硬盘主引导扇区的特征

- 文件大小为512字节（一个磁盘扇区的大小）
- 第510个字节为0x55，第511个字节为0xAA

```
// tools/sign.c
char buf[512];
    memset(buf, 0, sizeof(buf));
    FILE *ifp = fopen(argv[1], "rb");
    int size = fread(buf, 1, st.st_size, ifp);
    if (size != st.st_size) {
        fprintf(stderr, "read '%s' error, size is %d.\n", argv[1], size);
        return -1;
    }
    fclose(ifp);
    buf[510] = 0x55;
    buf[511] = 0xAA;
```

练习2：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。

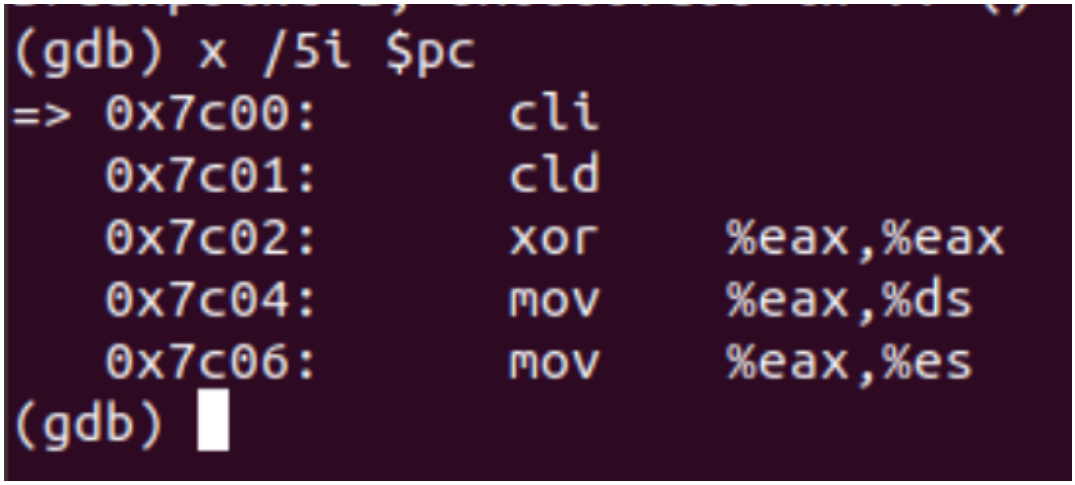
   cd进lab1-code文件夹后执行make debug命令，输入x /2i $pc可以查看附近两条汇编代码。

2. 在初始化位置0x7c00设置实地址断点,测试断点正常。

   - 修改gdbinit文件：去掉continue和原来的break语句

     ```
     file bin/kernel
     target remote :1234
     ```

   - 设置断点：b* 0x7c00
   - 执行c命令
   - 测试断点正常：使用x /5i $pc查看附近五条汇编代码

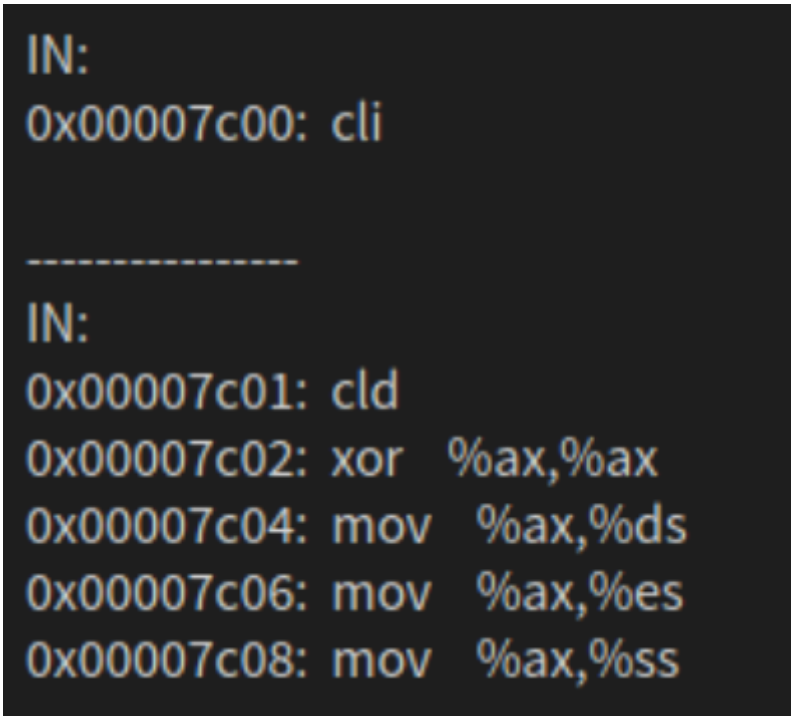


3. 从0x7c00开始跟踪代码运行,将单步跟踪反汇编得到的代码与bootasm.S和bootblock.asm进行比较。

   在makefile的debug代码中增加-d in_asm -D q.log 参数，将运行的汇编指令保存在q.log 中。

```
debug: $(UCOREIMG)
$(V)$(QEMU) -d in_asm -D q.log -S -s -parallel stdio -hda $< -serial null &
$(V)sleep 2
$(V)$(TERMINAL) -e "gdb -q -tui -x tools/gdbinit"
```

q.log



bootasm.S

bootblock.asm × | bootasm.S ×

```
selector
.set CR0_PE_ON,             0x1                     # protected mode enable flag

# start address should be 0:7c00, in real mode, the beginning address of the
running bootloader
.globl start
start:
.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
```

bootblock.asm

bootblock.asm × | bootasm.S ×

```
00007c00 <start>:

# start address should be 0:7c00, in real mode, the beginning address of the
running bootloader
.globl start
start:
.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
```

结论：执行的汇编代码、bootasm.S与bootblock.asm中的代码一样

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。
    - 修改gdbinit代码：在bootblock.o中的bootmain设置断点
    - 测试断点正常：使用x /5i $pc查看附近五条汇编代码

```
file obj/bootblock.o
set architecture i8086
target remote :1234
break bootmain
```

```
boot/bootmain.c
82          }
83      }
84
85      /* bootmain - the entry of bootloader */
86      void
87      bootmain(void) {
88          // read the 1st page off disk
89          readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);
90
91          // is this a valid ELF?
92          if (ELFHDR->e_magic != ELF_MAGIC) {
93              goto bad;
94          }
remote Thread 1 In: bootmain                                    L87   PC: 0x7d0d
Breakpoint 1, bootmain () at boot/bootmain.c:87
(gdb) x /5i $pc
=> 0x7d0d <bootmain>:   push   %ebp
   0x7d0e <bootmain+1>: xor    %ecx,%ecx
   0x7d10 <bootmain+3>: mov    $0x1000,%edx
   0x7d15 <bootmain+8>: mov    $0x10000,%eax
   0x7d1a <bootmain+13>:        mov    %esp,%ebp
(gdb)
```

练习3：分析bootloader进入保护模式的过程

- 实模式：程序用到的地址都是真实的物理地址，地址计算方式为16*段寄存器值+段内偏移地址，CPU寻址方式为寄存器寻址、立即数寻址、内存寻址（直接寻址、基址寻址、变址寻址、基址变址寻址）
- 保护模式：分段存储管理机制，涉及逻辑地址、段描述符（描述段的属性）、段描述符表（包含多个段描述符的"数组"）、段选择子（段寄存器，用于定位段描述符表中表项的索引）
- bootloader的作用：切换保护模式与段机制；从硬盘上读取kernel in ELF格式的ucore kernel（跟在MBR后面的扇区）并放到内存中；跳转到ucoreOS的入口点执行，将控制权移交给ucore OS

该部分代码主要对应bootasm.S源码

1. 关中断，数据寄存器清零

```
    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
```

2. 开启A20，将A20地址线置1，使用32根地址线，访问4G空间
    - 开启A20的作用：如果不打开A20地址线，对于32位地址第20位恒为零，系统只能访问奇数兆的空间。
    - A20的开启流程：
        - 读取0x64端口
        - 将0xd1写到0x64端口
        - 等待8042 input buffer为空，向其发送写数据的指令
        - 再次等待8042 input buffer为空，将0xdf发送至0x60，打开A20

```
    # Enable A20:
    #  For backwards compatibility with the earliest PCs, physical
    #  address line 20 is tied low, so that addresses higher than
    #  1MB wrap around to zero by default. This code undoes this.
seta20.1:
    inb $0x64, %al                                  # Wait for not busy(8042
input buffer empty).
    testb $0x2, %al
    jnz seta20.1

    movb $0xd1, %al                                 # 0xd1 -> port 0x64
    outb %al, $0x64                                 # 0xd1 means: write data
to 8042's P2 port

seta20.2:
    inb $0x64, %al                                  # Wait for not busy(8042
input buffer empty).
    testb $0x2, %al
    jnz seta20.2

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

```
    outb %al, $0x60                                     # 0xdf = 11011111, means
set P2's A20 bit(the 1 bit) to 1
```

3. 初始化GDT表

- gdt段：定义了段表中的空描述符(第一个总是空描述符)， 代码段描述符，数据段描述符。其中代码段和数据段是重合的，从0地址开始，与当前CS寄存器的值相同，确保进入保护模式后程序可以正常跳转
- gdtdesc段：定义了GDTR寄存器中应该有的值，其高32位是段表的起始地址(gdt)，低16位为段表的limit
- 段表初始化：lgdt gdtdesc将gdtdesc的值加载到GDT

```
lgdt gdtdesc
gdt:
    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
```

4. 设置系统寄存器CR0

- 系统寄存器CR0的第0位PE(Protection Enable)用于使能保护模式。设置为1时就可以开启保护模式

```
    movl %cr0, %eax
    orl $CR0_PE_ON, %eax
    movl %eax, %cr0
```

5. 通过长跳转更新cs的基地址实现从实模式进入保护模式

- 改变寻址方式后，使用之前的CS和IP已经不能正确访问到下一条要执行的指令了。因此，要正确访问到段表中的代码段描述符，应该将cs设置为段表中第一项的偏移量0x0008（段选择子的低三位为控制位，第二位为0表示访问GDT,低两位为零表示最高特权级）
- cs寄存器是不能直接设置的。它只能通过程序跳转指令，如CALL,RET, INT, LJMP指令来改变。因此，这里需要一个长跳转指令，来改变cs寄存器中的值，使之可以正确访问下一条要执行的指令
- $PROT_MODE_CSEG代表段表中代码段的偏移0x8, protcseg为下一条指令的偏移地址

```
    # Jump to next instruction, but in 32-bit code segment.
    # Switches processor into 32-bit mode.
    ljmp $PROT_MODE_CSEG, $protcseg
```

6. 设置段寄存器，建立堆栈，转到保护模式完成，call进入bootmain

```
protcseg:
    # Set up the protected-mode data segment registers
    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

    # Set up the stack pointer and call into C. The stack region is from 0--
start(0x7c00)
    movl $0x0, %ebp
    movl $start, %esp
    call bootmain
```

练习4：bootloader加载ELF格式的OS的过程

bootloader如何读取硬盘扇区的?

Bootloader的主要功能就是要读取硬盘扇区上的操作系统，把它加载到内存中，然后把控制权转交给操作系统。当前硬盘数据存储到硬盘扇区中，一个扇区大小为512B。读一个扇区的流程可以大致分为四个步骤：等待磁盘准备好，发出读取扇区的命令，等待磁盘准备好，把磁盘扇区数据读到指定内存。对应bootmain.c的代码：

1. 等待磁盘准备好：读取command寄存器，判断硬盘状态是否就绪
    - 端口号0x1F7：command寄存器，用于读取和写入命令，返回磁盘状态

```
/* waitdisk - wait for disk ready */
static void
waitdisk(void) {
    while ((inb(0x1F7) & 0xC0) != 0x40)
        /* do nothing */;
}
```

2. 发出读取扇区的命令：将要读取的硬盘扇区的信息先写入前面那些寄存器，然后通过command寄存器发出读取的命令

```
    outb(0x1F2, 1);                         // 读取扇区数目1
    outb(0x1F3, secno & 0xFF);              // 读取扇区编号
    outb(0x1F4, (secno >> 8) & 0xFF);       //发出读取扇区的命令
    outb(0x1F5, (secno >> 16) & 0xFF);
    outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
    outb(0x1F7, 0x20);                      // cmd 0x20 - read sectors
```

3. 再次等待磁盘准备好后把磁盘扇区数据读到指定内存
    - insl指令读入一个长字符序列，单位为4字节，SECTSIZE制定要读入的4字节的数量，所以要除4保证读入512字节

```
    // wait for disk to be ready
    waitdisk();
    // read a sector
    insl(0x1F0, dst, SECTSIZE / 4);
```

bootloader是如何加载ELF格式的OS?

ELF，即Executable and Linkable Format，为可执行和可链接格式。对应boostmain.c的代码，bootloader加载ELF的OS可以分为以下几个步骤：

1. 将OS从硬盘读入到内存中，并判断是否合法
    - 通过判断magic字段，可以确定这是否为一个合法的ELF文件

```
    // read the 1st page off disk
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);

    // is this a valid ELF?
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }
```

2. 通过ELF文件头，获得该OS的基本信息，并且将可执行代码从程序头标志的位置读入到内存中被程序头指定的地址
    - 若是合法的ELF文件，则可以通过程序头的偏移量(phoff)以及程序头的数量(phnum)来读到程序头，从而可以获得程序运行的更加具体的信息

```
    // load each program segment (ignores ph flags)
    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);
    }
```

3. 跳转到ELF文件标志的起始地址开始执行OS的指令，把CPU控制器转交给OS

```
    // call the entry point from the ELF header
    // note: does not return
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();
```

练习5：实现函数调用堆栈跟踪函数

- 需求：完成kdebug.c中函数print_stackframe的实现，可以通过函数print_stackframe来跟踪函数调用堆栈中记录的返回地址
    - 在函数的嵌套调用过程中，硬件会保存一些寄存器信息，以便于函数调用完成后返回被调用函数的下一条语句
    - 函数栈的构建过程：ebp为基址指针寄存器，esp为指向栈顶的堆栈指针寄存器。ebp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
- 核心代码

```
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]
      */
      // ebp是基址指针寄存器，eip是指令指针寄存器，这两个变量中储存的都是地址
      uint32_t ebp = read_ebp();
      uint32_t eip = read_eip();
      for(uint32_t i = 0; ebp && i < STACKFRAME_DEPTH; i++)
      {
          // 按规定格式打印堆栈信息
          cprintf("ebp:0x%08x eip:0x%08x args:", ebp, eip);
          // 用arguments接收ebp+2的地址，并输出arguments[0-3]
          uint32_t* args = (uint32_t*)ebp + 2 ;
          for(uint32_t j = 0; j < 4; j++)
              cprintf("0x%08x ", args[j]);
          cprintf("\n");
          // 栈底是高位地址，栈的方向是从高地址向低地址增长
          // 输出caller信息：eip指向异常指令的下一条指令
          print_debuginfo(eip-1);
          // 需要先设置eip后设置ebp，否则当ebp被修改后，eip就无法找到正确的位置
          // ebp指针指向的位置向上一个地址为上一个函数的eip
          eip = ((uint32_t*)ebp)[1];
          // ebp指针指向的位置存储的上一个ebp的地址
          ebp = ((uint32_t*)ebp)[0];
      }
}
```

- 运行结果：在命令行输入make qemu

```
Kernel executable memory footprint: 64KB
ebp:0x7b3800100bcc eip:0x00100bcc args:0x00010094 0x0010e950 0x00007b68 0x00100
0a2
    kern/debug/kdebug.c:306: print_stackframe+33
ebp:0x7b4800100f4d eip:0x00100bcc args:0x00000000 0x00000000 0x00000000 0x00100
08d
    kern/debug/kmonitor.c:125: mon_backtrace+23
ebp:0x7b68001000a2 eip:0x00100bcc args:0x00000000 0x00007b90 0xffff0000 0x00007
b94
    kern/init/init.c:48: grade_backtrace2+32
ebp:0x7b88001000d1 eip:0x00100bcc args:0x00000000 0xffff0000 0x00007bb4 0x00100
0e5
    kern/init/init.c:53: grade_backtrace1+37
ebp:0x7ba8001000f8 eip:0x00100bcc args:0x00000000 0x00100000 0xffff0000 0x00100
109
    kern/init/init.c:58: grade_backtrace0+29
ebp:0x7bc800100124 eip:0x00100bcc args:0x00000000 0x00000000 0x00000000 0x00103
7a4
    kern/init/init.c:63: grade_backtrace+37
ebp:0x7be800100066 eip:0x00100bcc args:0x00000000 0x00000000 0x00000000 0x00007
c4f
    kern/init/init.c:28: kern_init+101
ebp:0x7bf800007d6e eip:0x00100bcc args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa750
2a8
    <unknow>: -- 0x00007d6d --
```

练习6：完善中断初始化和处理

- 中断机制：让外设在需要操作系统处理外设相关事件的时候，能够“主动通知”操作系统，即打断操作系统和应用的正常执行，让操作系统完成外设的相关处理，然后在恢复操作系统和应用的正常执行
- 中断的三种类型
  - 异步中断/外部中断：由CPU外部设备引起的外部事件如I/O中断、时钟中断、控制台中断等是异步产生的（即产生的时刻不确定）
  - 同步中断/内部中断/异常：在CPU执行指令期间检测到不正常的或非法的条件(如除零错、地址访问越界)所引起的内部事件
  - 陷入中断/软中断/系统调用：在程序中使用请求系统服务的系统调用而引发的事件
- IDT gate descriptions
  - Interrupt Gate：调用Interrupt Gate时，Interrupt会被CPU自动禁止
  - Trap Gate：CPU不会禁止或打开中断
- 中断处理起始阶段
  - CPU执行完每条指令后，判断中断控制器中是否产生中断。如果存在中断，则取出对应的中断变量
  - CPU根据中断变量，到IDT中找到对应的中断描述符。通过获取到的中断描述符中的段选择子，从GDT中取出对应的段描述符。此时便获取到了中断服务例程的段基址与属性信息，跳转至该地址
  - CPU会根据CPL和中断服务例程的段描述符的DPL信息确认是否发生了特权级的转换。若发生了特权级的转换，这时CPU会从当前程序的TSS信息（该信息在内存中的起始地址存在TR寄存器中）里取得该程序的内核栈地址，即包括内核态的ss和esp的值，并立即将系统当前使用的栈切换成新的内核栈
  - 内核栈：CPU需要**开始保存当前被打断的程序的现场**（即一些寄存器的值），以便于将来恢复被打断的程序继续执行。这需要利用内核栈来保存相关现场信息，即依次压入当前被打断程序使用的eflags，cs，eip，errorCode（如果是有错误码的异常）信息
  - CPU利用中断服务例程的段描述符将其第一条指令的地址加载到cs和eip寄存器中，**开始执行中断服务例程**。这意味着先前的程序被暂停执行，中断服务程序正式开始工作
- 中断处理终止阶段：IRET指令
  - 程序执行这条iret指令时，首先会从内核栈里弹出先前保存的被打断的程序的现场信息，即eflags，cs，eip重新开始执行
  - 如果存在特权级转换（从内核态转换到用户态），则还需要从内核栈中弹出用户态栈的ss和esp，即栈也被切换回原先使用的用户栈
  - 如果此次处理的是带有错误码（errorCode）的异常，CPU在恢复先前程序的现场时，并不会弹出errorCode，需要要求相关的中断服务例程在调用iret返回之前添加出栈代码主动弹出errorCode

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？
   - 根据mmu.h中的表项结构，可以计算得到该表项的大小为`16+16+5+3+4+1+2+1+16=64bits`，即8字节
   - gd_ss是段选择子，gd_off_15_0和gd_off_31_16共同组成一个段内偏移地址，根据段选择子和段内偏移地址可以得到中断处理程序的地址

```
/* 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
};
```

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。

```
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!
      */
   // 声明中断入口
   extern uintptr_t __vectors[];
   for (int i = 0; i < (sizeof(idt) / sizeof(struct gatedesc)); i++)
       // 为中断设置内核态权限
       SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
   // 设置从用户态转为内核态的中断的特权级为DPL_USER
   SETGATE(idt[T_SYSCALL], 0, GD_KTEXT, __vectors[T_SYSCALL], DPL_USER);
   // 加载IDT
   lidt(&idt_pd);
}
```

3. 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500次时钟中断后，调用print_ticks子程序，向屏幕上打印一行文字”500 ticks”。

```
/* trap_dispatch - dispatch based on what type of trap occurred */
static void trap_dispatch(struct trapframe *tf) {
    char c;
    switch (tf->tf_trapno) {
    case IRQ_OFFSET + IRQ_TIMER:
        /* LAB1 YOUR CODE : STEP 3 */
        /* handle the timer interrupt */
        /* (1) After a timer interrupt, you should record this event using a
global variable (increase it), such as ticks in kern/driver/clock.c
         * (2) Every TICK_NUM cycle, you can print some info using a
funciton, such as print_ticks().
         * (3) Too Simple? Yes, I think so!
         */
        // 全局变量ticks定义于kern/driver/clock.c
        ticks++;
        if(ticks % TICK_NUM == 0)
            print_ticks();
        break;
    // .........
```

3. 实验感想

通过lab1，我对操作系统有了一个基本的认识，比如如何加载和启动一个操作系统，操作系统的代码结构是什么样的，编译调试操作系统的过程。此外，在实验过程中我也加深了对操作系统理论课或其他计算机课程所学知识的理解，比如函数堆栈、中断处理、保护模式下的内存管理、用户态内核态还有磁盘这类硬件层面等内容，看到这些或许在课本上比较抽象的概念是如何真正在代码中实现的，感慨“纸上得来终觉浅”。做这次实验还有一个收获是熟悉了linux系统的一些用法，打好基础准备做接下来的实验。