实验2 UCore启动过程-第2部分

个人信息

院系专业	年级	姓名	学号
软件工程	18级	梁允楷	18342055

Table of Contents

练习3 分析 bootloader 进入保护模式的过程

> BIOS 将通过读取硬盘主引导扇区到内存,并转跳到对应内存中的位置执行 bootloader。请分析 bootloader 是如何完成从实模式进入保护模式的。

CPU 从地址 0:0x7c00 以实模式进入主引导程序 `bootloader`，先执行 `boot/bootasm.S` 中的汇编指令。

1. 初始化环境

   初始化环境包括了关中断、复位方向寄存器、清零寄存器等行为。

```
.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
```

2. 打开 A20 Gate

   由于实模式下，只提供 1MB 的可寻址空间，超过 1MB 的地址按 1MB 求模访问地址。为了访问更高位的地址，引入了 A20 Gate，也就是第 21 根地址线进行管理。一般开机时，默认关闭 A20 Gate，打开 A20 Gate后，方可实现更高位的寻址。

   打开 A20 Gate，需要通过键盘控制器 8042，在其不繁忙的时候，先对 64h 端口发送 0xd1 指令，然后对 60h 发送 0xdf 指令。详细代码如下：

```
seta20.1:
    inb $0x64, %al
    testb $0x2, %al
    jnz seta20.1

    movb $0xd1, %al
    outb %al, $0x64

seta20.2:
    inb $0x64, %al
    testb $0x2, %al
    jnz seta20.2

    movb $0xdf, %al
    outb %al, $0x60
```

3. 装载全局描述符表（GDT）

   使用指令 `lgdt` 可以将 GDT 的入口地址加载到全局描述符表寄存器（GDTR）里，然后 CPU 就可以通过 GDTR 来访问 GDT。gdt 和 gdt 描述符在引导区也有相应的存储，可供直接装载。

```
lgdt gdtdesc

# omitted codes

gdt:
    SEG_NULLASM
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)
    SEG_ASM(STA_W, 0x0, 0xffffffff)

gdtdesc:
    .word 0x17
    .long gdt
```

4. 设置 CR0 寄存器的保护位（PE）

   CR0 寄存器的 0 号位 PE 是启用保护位（protection enable），将其置为 1，系统进入保护模式。

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

# omitted codes

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

5. 初始化中断

   接下来，通过长跳转更新 cs 寄存器的值，并且在保护模式下设置段寄存器的值，通过设置堆栈指针寄存器的值来建立堆栈。

```
.set PROT_MODE_CSEG,        0x8
.set PROT_MODE_DSEG,        0x10

# omitted codes

    ljmp $PROT_MODE_CSEG, $protcseg

.code32
protcseg:
    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
```

然后，执行指令 `call bootmain` 进入 `bootloader` 的主方法。CPU加电启动时，`IDTR.base` 被初始化为 0，往后对中断向量表继续初始化的操作将在练习6继续讨论。

练习4 分析 bootloader 加载 ELF 格式的 OS 的过程

> 通过阅读 bootmain.c,了解 bootloader 如何加载 ELF 文件。通过分析源代码和通过 qemu 来运行并调试 bootloader & OS。
>
> 1. bootloader 如何读取硬盘扇区的?
> 2. bootloader 是如何加载 ELF 格式的 OS?

bootloader 如何读取硬盘扇区的

1. `readsect` 函数

   我们可以在 `boot/bootmain.c` 第 43 行中找到函数 `readsect`，其作用是读取一个扇区内容。

   从这个函数我们可以了解到要读入扇区的内容，首先要等待磁盘进入空闲状态。磁盘的状态可以通过 IO 地址寄存器 0x1f7（状态和命令寄存器）来获得，详见 `waitdisk` 函数：当

`inb(0x1f7) & 0xc0 == 0x40` 时，磁盘进入空闲状态。

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

/* readsect - read a single sector at @secno into @dst */
static void readsect(void *dst, uint32_t secno) {
    // wait for disk to be ready
    waitdisk();
    // ...
```

指令 `outb(0x1F2, 1)` 用于表明读写的盘块数目为 1。接下来的 4 行代码，指明读取硬盘的扇区号，分别为：LBA 参数的 0-7 位、LBA 参数的 8-15 位、LBA 参数的 16-23 位和 LBA 参数的 24-27 位。剩余 0x1F6 的最高位用作指明当前通道的主从盘选择，0 为主盘，1 为从盘。

```
outb(0x1F3, secno & 0xFF);
outb(0x1F4, (secno >> 8) & 0xFF);
outb(0x1F5, (secno >> 16) & 0xFF);
outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);
```

然后的指令 `outb(0x1F7, 0x20)` 给硬盘发送了读取命令，在硬盘空闲之后即可从 IO 地址 0x1f0 读取数据内容到 `dst` 中。

```
outb(0x1F7, 0x20);      // cmd 0x20 - read sectors
// wait for disk to be ready
waitdisk();
// read a sector
insl(0x1F0, dst, SECTSIZE / 4);
```

2. `readseg` 函数

我们能在 `boot/bootmain.c` 的第 63 行找到 `readseg` 函数。它是在 `readsect` 的基础上进行封装的函数，能读取硬盘任意长度的内容。

```
static void readseg(uintptr_t va, uint32_t count, uint32_t offset) {
    uintptr_t end_va = va + count;

    // round down to sector boundary
    va -= offset % SECTSIZE;

    // translate from bytes to sectors; kernel starts at sector 1
    uint32_t secno = (offset / SECTSIZE) + 1;

    // If this is too slow, we could read lots of sectors at a time.
    // We'd write more to memory than asked, but it doesn't matter --
    // we load in increasing order.
    for (; va < end_va; va += SECTSIZE, secno ++) {
```

```
            readsect((void *)va, secno);
        }
    }
```

bootloader 是如何加载 ELF 格式的 OS

bootloader 加载 ELF 格式的 OS，有如下的步骤：

1. 读取 ELF 文件的头部

   上述的 readseg 函数可以读取任意位置任意大小的内容。我们借用这个函数读取 OS 镜像文件的第 1 页，大小为 8 个扇区，其中包含了 ELF 文件的头部信息。

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

2. 通过头部 e_magic 变量检查该 ELF 文件是否有效

   若不是有效的 ELF 文件，则程序进行错误处理，并进入一个死循环：

   ```
   // is this a valid ELF?
   if (ELFHDR->e_magic != ELF_MAGIC) {
       goto bad;
   }
   // omited codes
   bad:
       outw(0x8A00, 0x8A00);
       outw(0x8A00, 0x8E00);

       /* do nothing */
       while (1);
   ```

3. 加载 ELF 文件的程序段

   通过 elfheader 提供的 e_phoff 和 e_phnum 获得 program header table 的起始地址和所其中包含的条目数量。

   然后通过 program header table 可以访问各个 program header，从而读取各个程序段的内容。

   ```
   struct proghdr *ph, *eph;

   // 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);
   }
   ```

4. 执行 OS 内核

ELF 文件的头部信息中有执行内核的入口信息 `e_entry`，以此可以执行 OS 内核代码：

```
// 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

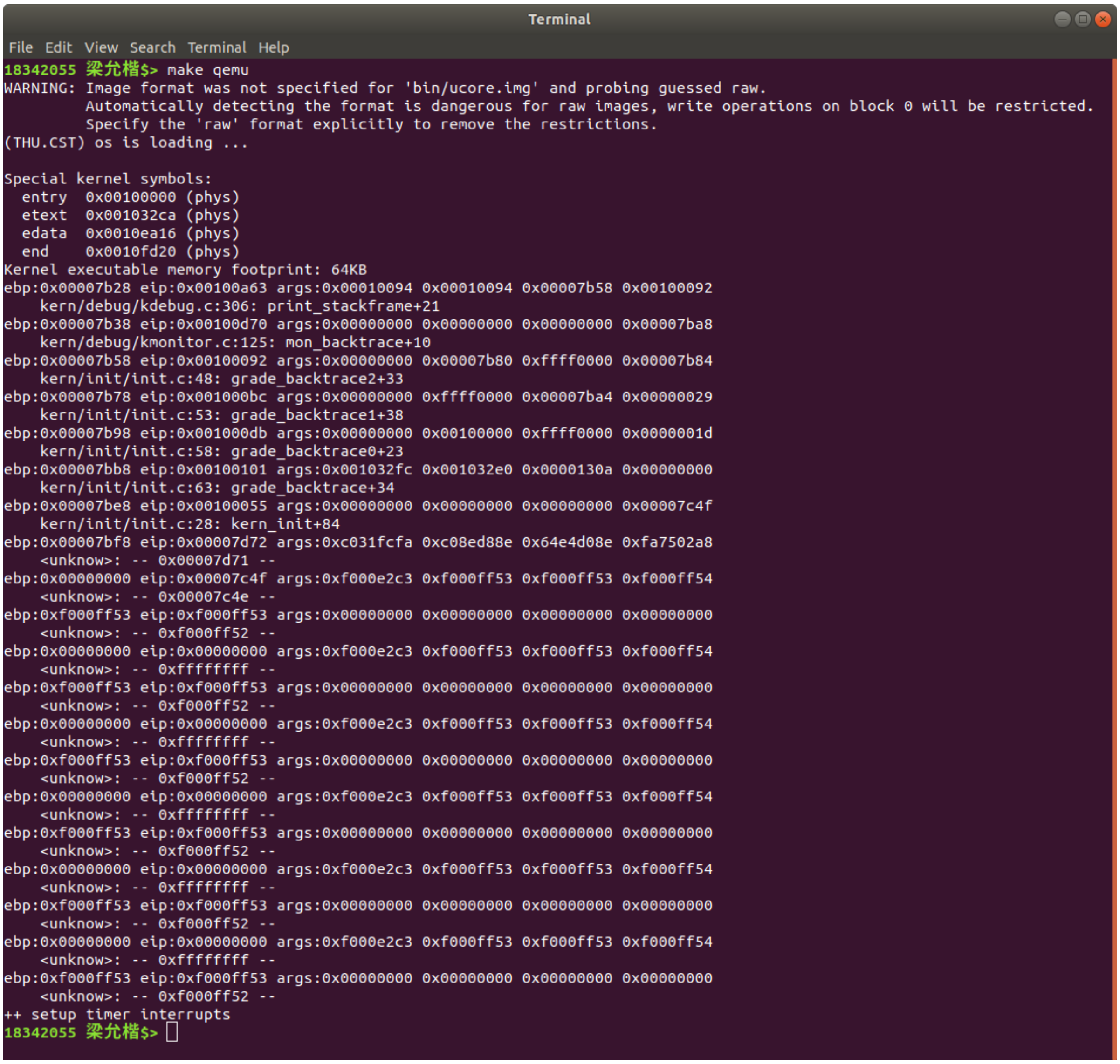
> 解释最后一行各个数值的含义。完成lab1编译后,查看 `lab1/obj/bootblock.asm`，了解 `bootloader` 源码与机器码的语句和地址等的对应关系；查看 `lab1/obj/kernel.asm`，了解 ucore OS 源码与机器码的语句和地址等的对应关系。
>
> 完成函数kern/debug/kdebug.c::print_stackframe的实现,提交改进后源代码包(可以编译执行),并在实验报告中简要说明实现过程,并写出对上述问题的回答。

实现 `print_stackframe` 函数

该函数按照项目所给的注释来进行操作，实现基本无问题。详细代码可参见 kedebug.c，或参见结尾附录 print_stackframe 函数实现。程序的大致实现过程为：

1. 获得当前 ebp 和 eip 寄存器的值
2. 进入循环：显示当前栈桢的相关信息，指令信息等内容
3. 通过 `eip = *(uint32_t *)(ebp + 4)` 和 `ebp = *(uint32_t *)ebp` 两条指令更新 `eip` 和 `ebp`，获取上一个栈桢的信息
4. 若获得足够的信息，则跳出循环；否则，返回步骤 2

所得结果如下，与题目所给的例子大致相同：



```
Terminal
File  Edit  View  Search  Terminal  Help
18342055 梁允楷$> make qemu
WARNING: Image format was not specified for 'bin/ucore.img' and probing guessed raw.
         Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0x00100000 (phys)
  etext  0x001032ca (phys)
  edata  0x0010ea16 (phys)
  end    0x0010fd20 (phys)
Kernel executable memory footprint: 64KB
ebp:0x00007b28 eip:0x00100a63 args:0x00010094 0x00010094 0x00007b58 0x00100092
    kern/debug/kdebug.c:306: print_stackframe+21
ebp:0x00007b38 eip:0x00100d70 args:0x00000000 0x00000000 0x00000000 0x00007ba8
    kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b58 eip:0x00100092 args:0x00000000 0x00007b80 0xffff0000 0x00007b84
    kern/init/init.c:48: grade_backtrace2+33
ebp:0x00007b78 eip:0x001000bc args:0x00000000 0xffff0000 0x00007ba4 0x00000029
    kern/init/init.c:53: grade_backtrace1+38
ebp:0x00007b98 eip:0x001000db args:0x00000000 0x00100000 0xffff0000 0x0000001d
    kern/init/init.c:58: grade_backtrace0+23
ebp:0x00007bb8 eip:0x00100101 args:0x001032fc 0x001032e0 0x0000130a 0x00000000
    kern/init/init.c:63: grade_backtrace+34
ebp:0x00007be8 eip:0x00100055 args:0x00000000 0x00000000 0x00000000 0x00007c4f
    kern/init/init.c:28: kern_init+84
ebp:0x00007bf8 eip:0x00007d72 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
    <unknow>: -- 0x00007d71 --
ebp:0x00000000 eip:0x00007c4f args:0xf000e2c3 0xf000ff53 0xf000ff53 0xf000ff54
    <unknow>: -- 0x00007c4e --
ebp:0xf000ff53 eip:0xf000ff53 args:0x00000000 0x00000000 0x00000000 0x00000000
    <unknow>: -- 0xf000ff52 --
ebp:0x00000000 eip:0x00000000 args:0xf000e2c3 0xf000ff53 0xf000ff53 0xf000ff54
    <unknow>: -- 0xffffffff --
ebp:0xf000ff53 eip:0xf000ff53 args:0x00000000 0x00000000 0x00000000 0x00000000
    <unknow>: -- 0xf000ff52 --
ebp:0x00000000 eip:0x00000000 args:0xf000e2c3 0xf000ff53 0xf000ff53 0xf000ff54
    <unknow>: -- 0xffffffff --
ebp:0xf000ff53 eip:0xf000ff53 args:0x00000000 0x00000000 0x00000000 0x00000000
    <unknow>: -- 0xf000ff52 --
ebp:0x00000000 eip:0x00000000 args:0xf000e2c3 0xf000ff53 0xf000ff53 0xf000ff54
    <unknow>: -- 0xffffffff --
ebp:0xf000ff53 eip:0xf000ff53 args:0x00000000 0x00000000 0x00000000 0x00000000
    <unknow>: -- 0xf000ff52 --
ebp:0x00000000 eip:0x00000000 args:0xf000e2c3 0xf000ff53 0xf000ff53 0xf000ff54
    <unknow>: -- 0xffffffff --
ebp:0xf000ff53 eip:0xf000ff53 args:0x00000000 0x00000000 0x00000000 0x00000000
    <unknow>: -- 0xf000ff52 --
ebp:0x00000000 eip:0x00000000 args:0xf000e2c3 0xf000ff53 0xf000ff53 0xf000ff54
    <unknow>: -- 0xffffffff --
ebp:0xf000ff53 eip:0xf000ff53 args:0x00000000 0x00000000 0x00000000 0x00000000
    <unknow>: -- 0xf000ff52 --
++ setup timer interrupts
18342055 梁允楷$>
```

解释最后一行各个数值的含义

此处使用题目所提供的数据进行说明：

```
ebp:0x00007bf8 eip:0x00007d73 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
    <unknow>: -- 0x00007d72 –
```

在汇编语言中调用函数，需要进行恢复现场等一系列操作，为此，调用函数时，有一系列的规则，并引入 ebp 寄存器方便我们解决这一问题。

ebp 寄存器与 ss 段寄存器组合构成 `ss:ebp`。它指向的位置在堆栈中存储着主调函数的 ebp 的值。该 ebp 值的上方保存着主调函数的相关信息，如：`ss:ebp+4` 指向主调函数调用时的 eip，若被调函数需要使用参数则 `ss:ebp+8` 指向被调函数用到的第一个参数。该 ebp 值的下方保存着被调函数的相关信息，如局部变量等信息。

此处最后一行对应着堆栈的最深一层，即最初使用堆栈的函数 `bootmain`。在练习 3 的步骤 5 处我们已经对 ebp 和 esp 寄存器进行了赋值，`bootmain` 函数所使用的堆栈结构位于 0:0x7c00 处。因此，记录了一次返回地址和一次 ebp 寄存器后，`bootmain` 函数所使用的栈桢的基址为 0x7bf8。调用函数的指令存储的位置为 0x7d73，由于bootmain 不需要使用参数，因此后面的参数：0xc031fcfa、0xc08ed88e、0x64e4d08e、0xfa7502a8 为无用参数。

练习6 完善中断初始化和处理

> 请完成编码工作和回答如下问题：
>
> 1. 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？
> 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。注意除了系统调用中断(T_SYSCALL)以外，其它中断均使用中断门描述符，权限为内核态权限;而系统调用中断使用异常，权限为陷阱门描述符。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。
> 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 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”。
>
> 完成问题 2 和 3 要求的部分代码后，运行整个系统，可以看到大约每 1 秒会输出一次”100 ticks”，而按下的键也会在屏幕上显示。

关于中断向量表的描述

中断向量表（IDT）是一个 8 字节的描述符数组。因此，中断向量表一个表项占 8 字节。其中，第 16～47 位（共 32 位）表示中断处理代码的入口。

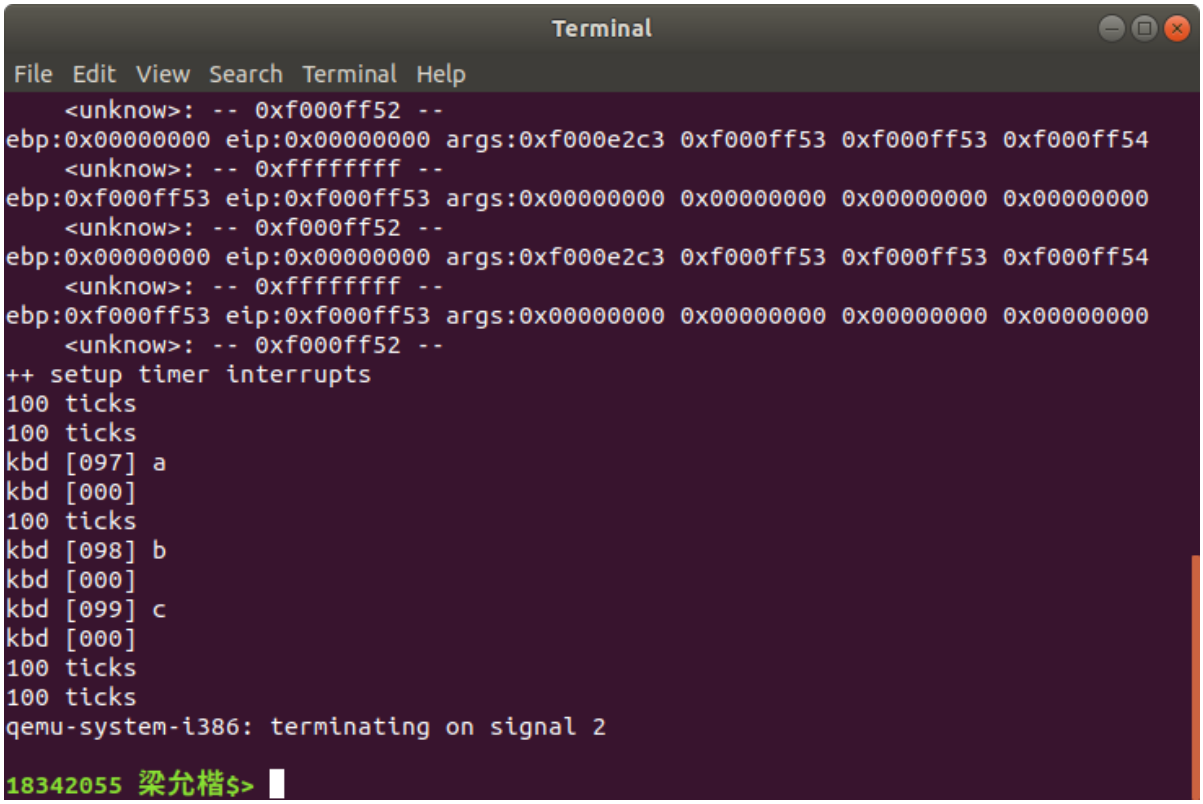
对中断向量表的初始化

根据项目提供的注释，在 kern/trap/vectors.S 中提供了 __vectors 的相关定义。使用 __vectors 和宏函数 SETGATE 可以对中断向量表 idt 进行初始化。SETGATE 中的参数可以根据课件中给的提示进行设置。值得注意的是，SETGATE 中的 sel 参数需要在 kern/mm/memlayout.h 中找到全局代码段选择子对应的数值 GD_KTEXT。最后还要调用 lidt 指令，传递相应的基址和限制记录中断向量表的信息。详细代码可见结尾附录 idt_init 函数的实现或 trap.c 文件。

> 除了系统调用中断(T_SYSCALL)使用陷阱门描述符且权限为用户态权限以外,其它中断均使用特权级(DPL)为0的中断门描述符,权限为内核态权限。

对时钟中断部分的完善处理

根据项目的注释，我们对时钟中断进行了自定义处理：每经历 100 次时钟中断，输出一次 100 ticks。代码比较简单，同样具体代码见详细代码可见结尾附录 trap_dispatch 函数时钟中断部分的实现或 trap.c 文件。

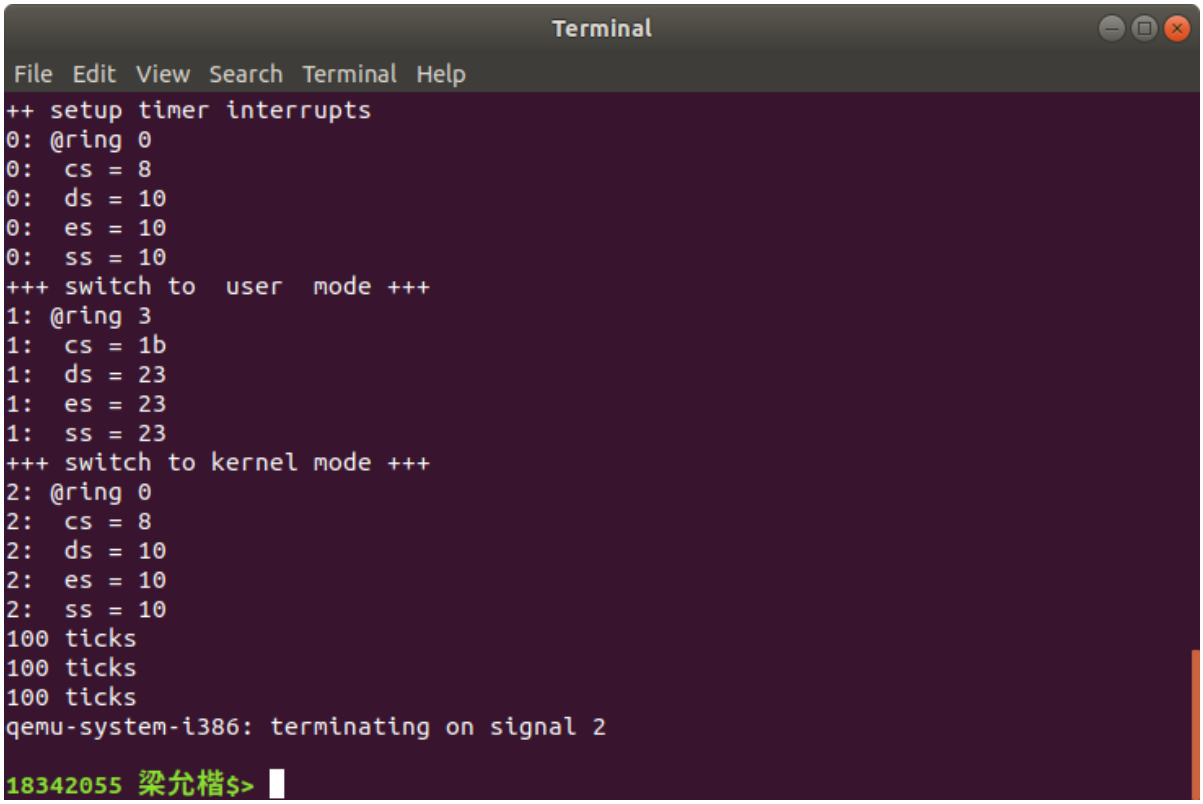


扩展练习 实现内核态与用户态相互转换

> 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。

经过反复查阅网上资料，大致上了解实现用户态与内核态转换的方法。核心思想通过把所有的寄存器的值 push 入栈中，通过一个陷入帧对栈里保存的值进行修改。最后通过 pop 修改寄存器的值。实现对 cs、ds、es、ss、eflags 等寄存器的值进行修改，进入用户态/核心态。

详细代码参见附录或者 trap.c 文件和 init.c 文件。



实验心得

本次实验从机器启动到操作系统运行，逐步介绍整个系统运行的过程。从 BIOS 启动、bootloader 启动、最后到操作系统启动，初始化 gdt、idt 等。从第一条指令执行开始，对计算机系统有了更多深入的了解。期间深深的感受到 intel 由于历史包袱所带来深深的不便。为了前向兼容，intel 作出了相当多的设计，如：实模式、寄存器设计等。同时，通过对操作系统的引导，加强了多方面知识的学习。比如，对三种地址空间，反复进行回味；了解硬盘访问 LBA 模式是如何传参，进行对应功能的操作；如何对硬件直接编程也有了相应的了解。通过自主学习，也认识到内核态和用户态进行转换的大致思路。通过这次实验，了解了很多东西，也意识到操作系统的搭建是一个不断搭积木的过程，随着我们的零件逐渐增多，我们进行操作系统的搭建也慢慢容易起来。

附录

kern/debug/kdebug.c 中的 print_stackframe 函数

```
  void print_stackframe(void) {
      uint32_t ebp = read_ebp();
      uint32_t eip = read_eip();
      for (int i = 0; i < STACKFRAME_DEPTH; i++) {
          cprintf("ebp:0x%08x eip:0x%08x ", ebp, eip);
          uint32_t *args = (uint32_t *) ebp + 2;
          for (int ofst = 0; ofst < 4; ofst++) {
              if (ofst == 0) cprintf("args:0x%08x", args[ofst]);
              else cprintf(" 0x%08x", args[ofst]);
          }
          cprintf("\n");
          print_debuginfo(eip - 1);
          eip = *(uint32_t *)(ebp + 4);
```

```
            ebp = *(uint32_t *)ebp;
        }
    }
```

kern/trap/trap.c 中的 idt_init 函数

```
/* idt_init - initialize IDT to each of the entry points in kern/trap/vectors.S */
void idt_init(void) {
    extern uintptr_t __vectors[];
    for (int i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i++) {
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
    SETGATE(idt[T_SYSCALL], 1, GD_KTEXT, __vectors[T_SYSCALL], DPL_USER);
    lidt(&idt_pd);
}
```

kern/trap/trap.c 中的 trap_dispatch 函数时钟中断部分

```
case IRQ_OFFSET + IRQ_TIMER:
    /* LAB1 YOUR CODE : STEP 3 */
    ticks++;
    if (ticks % TICK_NUM == 0) {
        print_ticks();
    }
    break;
```

拓展练习相关代码

- kern/init/init.c 相关改动

  在 kern_init 函数末尾增加一句：lab1_switch_test();。

  完成函数 lab1_switch_to_user 和 lab1_switch_to_kernel：

  ```
  static void
  lab1_switch_to_user(void) {
      //LAB1 CHALLENGE 1 : TODO
      __asm__ __volatile__(
          "sub $8, %%esp\n"
          "int %0\n"
          "movl %%ebp, %%esp"
          :
          : "i"(T_SWITCH_TOU)
      );
  }

  static void
  lab1_switch_to_kernel(void) {
      //LAB1 CHALLENGE 1 :  TODO
      __asm__ __volatile__(
          "int %0\n"
  ```

```
            "movl %%ebp, %%esp"
            :
            : "i"(T_SWITCH_TOK)
        );
    }
```

- kern/trap/trap.c 对 trap_dispatch 函数的改动

```
    //LAB1 CHALLENGE 1 : YOUR CODE you should modify below codes.
    case T_SWITCH_TOU:
        tf->tf_cs = USER_CS;
        tf->tf_ds = USER_DS;
        tf->tf_es = USER_DS;
        tf->tf_ss = USER_DS;

        tf->tf_eflags |= FL_IOPL_MASK;
        break;
    case T_SWITCH_TOK:
        tf->tf_cs = KERNEL_CS;
        tf->tf_ds = KERNEL_DS;
        tf->tf_es = KERNEL_DS;

        tf->tf_eflags &= ~FL_IOPL_MASK;
        break;
```