操作系统实验一

学号：2013921
姓名：周延霖
专业：信息安全

一、遇到的BUG和解决

> Q：在gdbinit文件中加入set architechture i8086和break *0xffff0两条命令，尝试查看BIOS执行的第一条指令时，gdb调试失败，没有进入命令行模式，显然gdbinit文件修改得并不正确。到底应当给出什么命令?

A：按照指导书中写法，将gdbinit文件内容改成只有那两行命令即可。如果按照原来gdbinit文件里的写法，在kern_init函数入口处停下，地址则是0x100000H，但kern_init函数本身用于加载操作系统内核，与BIOS无关。

> Q：A20的“线号乌龙“

A：一开始查资料的时候一直看到A20是在第20根线上完成的兼容开关设置，我很疑惑，原来就是20根线，现在却在第20根线上做开关，那不是占用了原本那根线的传输功能么？后来再找了一些资料求证，可以合理认为，这里的”第20根“是从”0“开始编号计数的。开关是放在了实际增加的第1根线上，用于控制后面的地址线传输是否有效。

> Q：bootmain函数最后跳去了哪里？为什么要跳?

A：使用gdb查看代码，发现跳到了0x00100000的位置，即kern_init的函数入口处，也就是操作系统在内存中被载入的位置。

> Q：为什么没有bootmain到kern_init的函数栈帧切换?

A：查看bootmain的最后， ((void (*)(void))(ELFHDR->e_entry & 0xFFFFFF))();可见是使用指针跳转到了kern_init的入口地址，因为不存在函数堆栈的栈帧切换，那么在查看运行的堆栈结果时，自然就查看不到bootmain到kern_init的栈帧切换了。

> Q：为什么init.c中switch_to_user要先把esp-8？而switch_to_kernel不用？或者，换个问法：在内核态切换到用户态需要修改ss，为什么用户态到内核态不用修改ss?

A：在中断返回时，先弹出eip和cs，再根据cs判断会返回到什么特权级。如果要返回到更低的特权级，会弹出ss和esp。用户态的cs=1b=00011011，CPL=DPL=RPL=3，内核态的cs=08=00001000，CPL=DPL=RPL=0。所以内核态到用户态的过程中，是从CPL=0切换到DPL=3，弹出esp和ss，这里弹出的ss是user_ss，存放在ss中，实现内核态到用户态的切换。但是在用户态到内核态的过程中，是从CPL=3切换到DPL=0，是返回到一个更高的特权级，所以本身没有esp和ss被弹出，又因为在用户态下中断，TSS会帮助把用户态的esp和ss压栈，并把当前的ss切换为内核态的ss。刚进入中断的时候我们的ss已经变成内核态的ss了，自然不用修改。

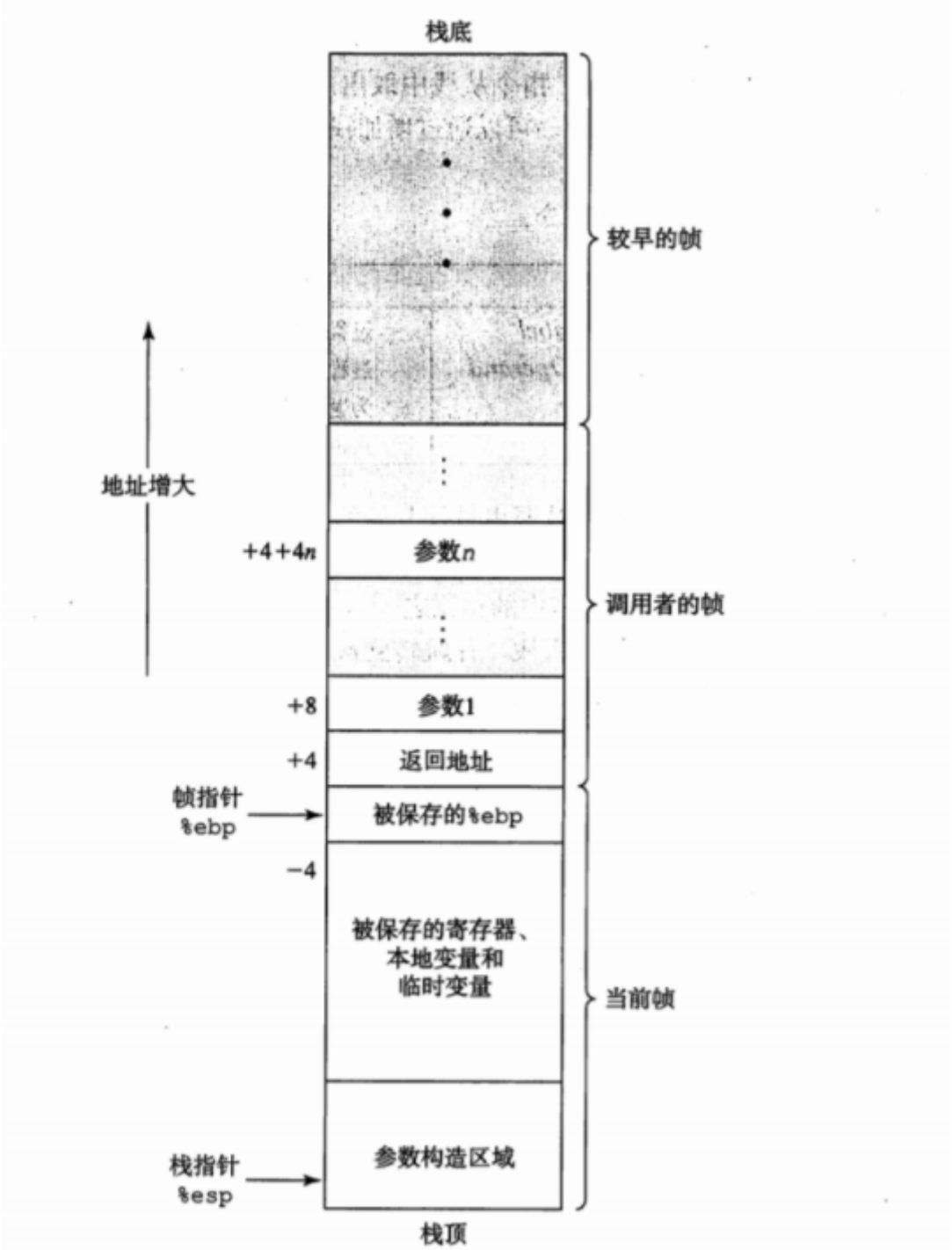
二、练习5：实现函数调用堆栈跟踪函数(需要编程)

在lab1/kern/debug目录下找到kdebug.c,打开以后发现源文件中已经有一个print_stackframe函数了(虽然里面啥也没有)，要做的就是往里面添加代码，让这个函数能实现我们需要的功能。以下是初始状态的代码。

```
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 (uint32_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]
      */
}
```

为了实现函数调用堆栈跟踪函数，需要先了解函数调用栈的原理。

函数调用时，自栈顶(低地址)到栈底(高地址)的情况如下图所示



栈帧结构（栈用来传递参数、存储返回信息、保存寄存器，以及本地存储）

esp和ebp两个指针是其中最关键的部分，只要掌握了ebp和esp的位置，就能很容易理解函数调用过程了。根据这个图，有这样几个信息

- ss[ebp]指向上一层的ebp
- ss[ebp-4]指向局部变量
- ss[ebp+4]指向返回地址
- ss[ebp+4+4n]指向第n个参数

之后就可以着手实现堆栈跟踪函数了。首先我们知道在bootasm.S中将esp设置为0x7c00，ebp设置为0，就调用了bootmain函数。call指令会依次执行以下命令:push返回地址，push这一层的ebp，然后把现在的esp赋值给ebp。在执行完call之后，这个ebp指向了0x7bf8(0x7c00-4-4)。

实现之前看了一下源代码中的注释，突然惊讶的发现注释已经非常贴心的教你怎么写了，那就按着这个注释一步一步来，写出如下的结果：

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

之后在lab1目录下执行命令 $ make qemu，得到如下的输出

```
......
Kernel executable memory footprint: 64KB
ebp=: 0x00007b28 | eip=: 0x00100a63 | args=: 0x00010094 0x00010094
0x00007b58 0x00100092
    kern/debug/kdebug.c:307: print_stackframe+21
ebp=: 0x00007b38 | eip=: 0x00100d4d | 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=: 0x001032dc 0x001032c0
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 --
......
```

最后一行中给出了ebp，eip和args三个参数，其具体意义为

- **ebp=: 0x00007bf8** 是跳转到bootmain
- **eip=: 0x00007d72** 是从bootasm.s跳转到bootmain前的地址，也就是bootmain的返回地址。
- **args=: 0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8** 通常状态下，args存放的四个dword是对应4个输入参数的值。但是再最底层处，即7c00往后增加的地址处，那里是bootloader的代码段，所以最后的args其实是bootloader指令的前十六个字节，下面这个例子就能很好的说明情况

```
# bootloader前三条指令对应的机器码
7c00:   cli             fa
7c01:   cld             fc
7c02:   xor    %eax,%eax    31 c0
# 由于是小端字节序，所以存储为 c0 31 fc fa
```

扩展练习 Challenge1(需要编程)

首先为了完成特权级转换，需要了解这些知识：

- int iret在不同情况下的执行步骤
- 特权级检查

阅读实验指导书，kern/init/init.c和kern/trap/trap.c文件，需要完成以下四个内容：

- kern/init/init.c 中的 switch_to_user
- kern/init/init.c 中的 switch_to_kernel
- kern/trap/trap.c 中的 case T_SWITCH_TOU #to user
- kern/trap/trap.c 中的 case T_SWITCH_TOK #to kernel

因为在调用关系中是 init.c调用trap.c，所以先从init.c中入手。

先来看switch_to_user。很好，什么也没有，满足我对于难度的要求。

```
static void
lab1_switch_to_user(void) {
    //LAB1 CHALLENGE 1 : TODO
}
```

没有东西只能白手起家，还能咋地。先把写好的代码贴出来，之后再进行详细的解释

```
//  init.c
static void
lab1_switch_to_user(void) {
    //LAB1 CHALLENGE 1 : TODO
    asm volatile(
            "sub $0x8,%%esp \n"
            "int %0 \n"
            "movl %%ebp, %%esp \n"
```

```
            :
            :"i"(T_SWITCH_TOU)
        );
}
static void
lab1_switch_to_kernel(void) {
    //LAB1 CHALLENGE 1 :  TODO
    asm volatile(
            "int %0 \n"
            "movl %%ebp, %%esp \n"
            :
            :"i"(T_SWITCH_TOK)
        );
}
```

```
// trap.c
    case T_SWITCH_TOU:
        if(tf->tf_cs != USER_CS)    //检查是不是用户态，不是就操作
        {
                cprintf("...to user\n");
                // 设置用户态对应的cs,ds,es,ss四个寄存器
                tf->tf_cs = USER_CS;
                tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
                // 为用户态带来可以I/O的快乐
                tf->tf_eflags |= FL_IOPL_MASK;
        }
        break;

    case T_SWITCH_TOK:
        if(tf->tf_cs != KERNEL_CS)  //检查是不是内核态，不是就操作
        {
                cprintf("...to kernel\n");
                // 设置内核态对应的cs,ds,es三个寄存器
                tf->tf_cs = KERNEL_CS;
                tf->tf_ds = tf->tf_es = KERNEL_DS;
                // 剥夺用户态可以使用I/O的快乐
                tf->tf_eflags &= ~FL_IOPL_MASK;
        }
        break;
```

一开始我以为user_to_kernel和kernel_to_user应该没有什么区别，但这个challenge1不愧是个challenge。其中的区别在中断发生的压栈状况有关系。

中断可以发生在任何一个特权级别下，但是不同的特权级处理器使用的栈不同，如果涉及到特权级的变化，需要对SS和ESP寄存器进行压栈。性质如下：

- 当低特权级向高特权级切换的压栈(**用户态到内核态**)

  需要判断是否能访问这个目标段描述符，要做的就是将找到中断描述符时的CPL与目标段描述符的DPL进行比较。当CPL特权级比DPL低(CPL>DPL)时，要往高特权级栈转移，也就是说要恢复旧栈，因此处

理器临时保存旧栈的SS和ESP，然后加载新的特权级和DPL相同的段到SS和ESP中，把旧栈的SS和ESP压入新栈

- 当无特权级转化时的压栈(**内核态到用户态**)

  理论上来说从内核态到用户态也需要对栈进行切换，不过在lab1中并没有完整实现对物理内存的管理，而GDT中的每一个段除了对特权级的要求以外都一样，所以只需要修改一下权限就可以实现了。这也导致这个时候不会压栈，我们需要手动压栈(体现在lab1_switch_to_user中的sub $0x8,%%esp)。

不过在trap.c中的实现比较雷同，把对应的tf指针修改为对应态的内容就行。

扩展练习 Challenge2(需要编程)

> 用键盘实现用户模式内核模式切换。具体目标是：“键盘输入3时切换到用户模式，键盘输入0时切换到内核模式”。 基本思路是借鉴软中断(syscall功能)的代码，并且把trap.c中软中断处理的设置语句拿过来。

在Challenge1中其实已经实现了用户模式和内核模式的相互切换，所需要的只是增加一个用键盘输入来控制切换的功能。找到控制状态切换的trap.c文件中的trap_dispatch函数。他已经很贴心的给我们写了一个case IRQ_OFFSET + IRQ_KBD:(键盘输入情况)

很自然的能够想到，用**if-else**结构就可以实现一个控制。那么到此编程的思路已经十分明确了。直接贴出代码：

```
case IRQ_OFFSET + IRQ_KBD:
        c = cons_getc();
        cprintf("kbd [%03d] %c\n", c, c);
        if(c == '0' && (tf->tf_cs & 3) != 0)
        {
                cprintf("Input 0......switch to kernel\n");
                tf->tf_cs = KERNEL_CS;
                tf->tf_ds = tf->tf_es = KERNEL_DS;
                tf->tf_eflags &= ~FL_IOPL_MASK;
        }
        else if (c == '3' && (tf->tf_cs & 3) != 3)
        {
                cprintf("Input 3......switch to user\n");
                tf->tf_cs = USER_CS;
                tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
                tf->tf_eflags |= FL_IOPL_MASK;
        }
        break;
```