扩展练习 Challenge（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。需写出详细的设计和分析报告。）

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();
}
```

switch_to_*函数建议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。

答：

在文件 kern/init/init.c 中，在练习 6 的使能中断命令之后，调用问题出给出的函数 switch_test，函数的作用是打印当前和切换到用户模式以及内核模式后 CS、SS、DS 等寄存器的状态。

其中切换到用户模式调用了函数 lab1_switch_to_user，写出它的代码如下：

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

这段代码完全由扩展 GCC 内联汇编来实现，volatile 保证 asm 指令不被删除、移动或组合，内联汇编的基本语法规则如下：

```
asm [ volatile ] (
    assembler template
```

```
    [ : output operands ]                   /* 可选的 */
    [ : input operands  ]                   /* 可选的 */
    [ : list of clobbered registers ]   /* 可选的 */
    );
```

汇编指令中的操作数，可能出现%0、%1 等，若命令共涉及 n 个操作数，则第 1 个输出操作数（the first output operand）被编号为 0，第 2 个输出操作数编号为 1，依次类推，最后 1 个输入操作数（the last input operand）则被编号为 n-1。list of clobbered registers 用于列出指令中涉及到的且没出现在 output operands 字段及 input operands 字段的那些寄存器。若寄存器被列入 clobber-list，则等于是告诉 gcc，这些寄存器可能会被内联汇编命令改写。因此，执行内联汇编的过程中，这些寄存器就不会被 gcc 分配给其它进程或命令使用。[1]

`int` 指令进行下面一些步骤：

1. 从 IDT 中获得第 n 个描述符，n 就是 `int` 的参数。
2. 检查`%cs` 的域 CPL<=DPL，DPL 是描述符中记录的特权级。
3. 如果目标段选择符的 PL<CPL，就在 CPU 内部的寄存器中保存`%esp` 和`%ss` 的值。（这里的目标段选择符我不太清楚是什么，我觉得可能是相应的 GDT 表项里特区级或者就是这个中断描述符里的 CS 所记录的特权级）
4. 从一个任务段描述符中加载`%ss` 和`%esp`。（这里的任务段就是 TSS，一般设定为每个 CPU 一个，且在固定的段表项）
5. 将`%ss` 压栈。
6. 将`%esp` 压栈。
7. 将`%eflags` 压栈。
8. 将`%cs` 压栈。
9. 将`%eip` 压栈。
10. 清除`%eflags` 的一些位。
11. 设置`%cs` 和`%eip` 为描述符中的值。

如果步骤 3 不为真的话，3、4、5、6 这几个步骤都是不会执行的。因此，对于没有发生特权级转换的中断，其实是没有栈切换的。一直都是用同一个栈在处理中断。

切换到用户态过程中中第一行汇编代码 `sub $0x8, %esp` 的的作用是预留出 8 个字节存放 `iret` 的返回，这是由于切换特权级时，`iret` 指令会额外弹出 `SS` 和 `ESP`，但调用中断时并未产生特权级切换，因此并未压入对应 `SS` 和 `ESP`。需要预先留出空间防止代码出错。

第二行汇编代码 `int %0` 是调用切换至用户态的中断 `T_SWITCH_TOU`，该中断号已经在 kern/trap/trap.c 以及 kern/trap/vectors.S 中定义，这里直接进行调用，根据练习 6 的实验报告可知调用中断后会间接进入 `trap_dispatch` 函数，进而执行如下代码：

```
    //LAB1 CHALLENGE 1 : YOUR CODE you should modify below
codes.
    case T_SWITCH_TOU:
```

[1] https://blog.csdn.net/wdjjwb/article/details/77239612

```
        if (tf->tf_cs != USER_CS) {
            switchk2u = *tf;    //拷贝一份 tf 所指内容到新的位置
            switchk2u.tf_cs = USER_CS;  //用户态代码段
            switchk2u.tf_ds = switchk2u.tf_es =
switchk2u.tf_ss = USER_DS;  //用户态数据段
            switchk2u.tf_esp = (uint32_t)tf + sizeof(struct
trapframe) - 8;

            // set eflags, make sure ucore can use io under
user mode.
            // if CPL > IOPL, then cpu will generate a general
protection.
            switchk2u.tf_eflags |= FL_IOPL_MASK;

            // set temporary stack
            // then iret will jump to the right stack
            *((uint32_t *)tf - 1) = (uint32_t)&switchk2u;
//使用新栈恢复
        }
        break;
```

trap/trap.h 中结构体 `trapframe` 的定义如下：

```
50  /* registers as pushed by pushal */
51  struct pushregs {
52      uint32_t reg_edi;
53      uint32_t reg_esi;
54      uint32_t reg_ebp;
55      uint32_t reg_oesp;          /* Useless */
56      uint32_t reg_ebx;
57      uint32_t reg_edx;
58      uint32_t reg_ecx;
59      uint32_t reg_eax;
60  };
61
62  struct trapframe {
63      struct pushregs tf_regs;
64      uint16_t tf_gs;
65      uint16_t tf_padding0;
```

```
66      uint16_t tf_fs;
67      uint16_t tf_padding1;
68      uint16_t tf_es;
69      uint16_t tf_padding2;
70      uint16_t tf_ds;
71      uint16_t tf_padding3;
72      uint32_t tf_trapno;
73      /* below here defined by x86 hardware */
74      uint32_t tf_err;
75      uintptr_t tf_eip;
76      uint16_t tf_cs;
77      uint16_t tf_padding4;
78      uint32_t tf_eflags;
79      /* below here only when crossing rings, such as from
    user to kernel */
80      uintptr_t tf_esp;
81      uint16_t tf_ss;
82      uint16_t tf_padding5;
83  } __attribute__((packed));
```

其中第 74-78 行是在硬件产生中断之后硬件 CPU 自动保存的一些信息；第 63-72 行是软件保存的信息，`pushregs` 中的寄存器都是 `pushal` 中需要压入栈的所有寄存器；第 80-82 行考虑的是将来有可能出现从用户态产生中断会切换到内核态，那么就会多保存一些信息如用户态的栈（ESP、SS）。对于 x86 而言，用户态一般我们设置在特权级 3，而内核态设置在特权级 0。有了这个数据结构后，我们就可以在中断后获取中断的信息，并将它传给 ISR，ISR 会根据传入的 `trapframe` 来进行相应的操作。[2]

如果这时候的特权级不是 USER，则创建另一个栈（用户栈），通过 tf_esp 保存内核态的 tf 地址，改变新建栈寄存器的 DPL。同时将新建栈的地址放入压入到 ESP 处，弹出 ESP 时进入了新建栈，完成转换。

这里将 Eflags 的 IOPL 位设置为 3 是因为用户态返回后要进行 `cprintf` 调用，这个函数使用了 `in` 和 `out` 指令。但是如果不设置好 Eflags 的值的话，在用户态执行这两条指令是会产生 13 号中断错误。

整个过程如下图所示：

[2] https://www.jianshu.com/p/94fec16c5252

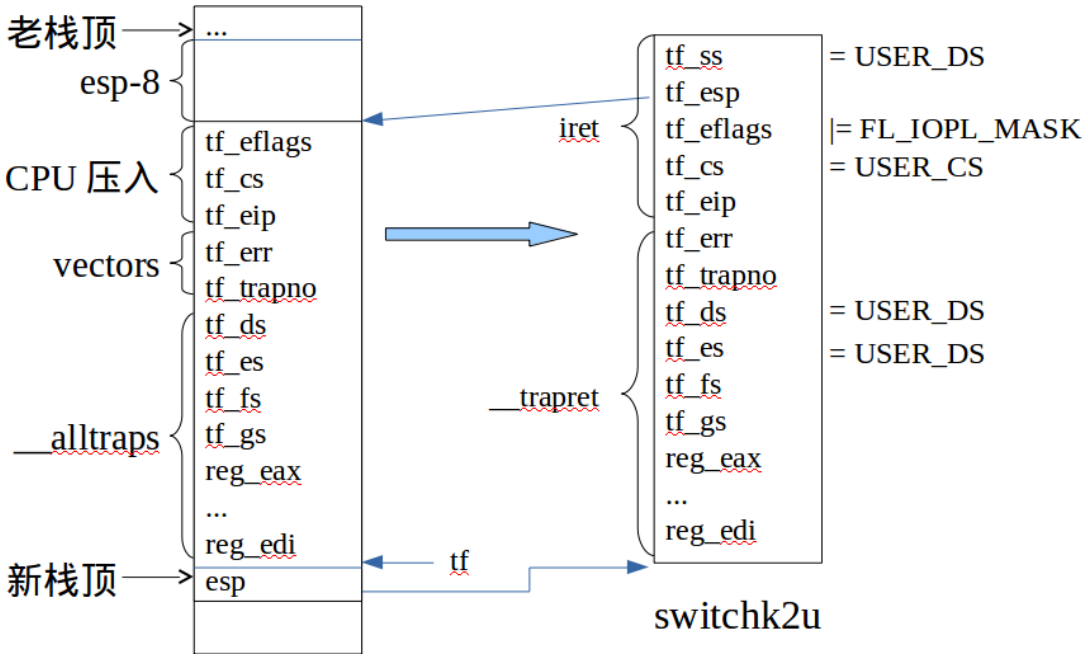

图 1 切换到用户态的栈情况

由于 `int` 指令和 `iret` 指令是一对，`iret` 指令的动作也是类似的。其指令的步骤如下：

1. 将`%eip` 弹栈。
2. 将`%cs` 弹栈。
3. 将`%eflags` 弹栈。
4. 将`%esp` 弹栈。
5. 将`%ss` 弹栈。

和前面类似，从用户态返回内核态是一个相反的过程，通过获得 tf_esp 来寻找到前一次进入用户态的内核态的代码所在处，通过计算得到图 2 中新栈顶的位置，同时把该栈放入 ESP 处，为下一次的操作转换做准备。[3]

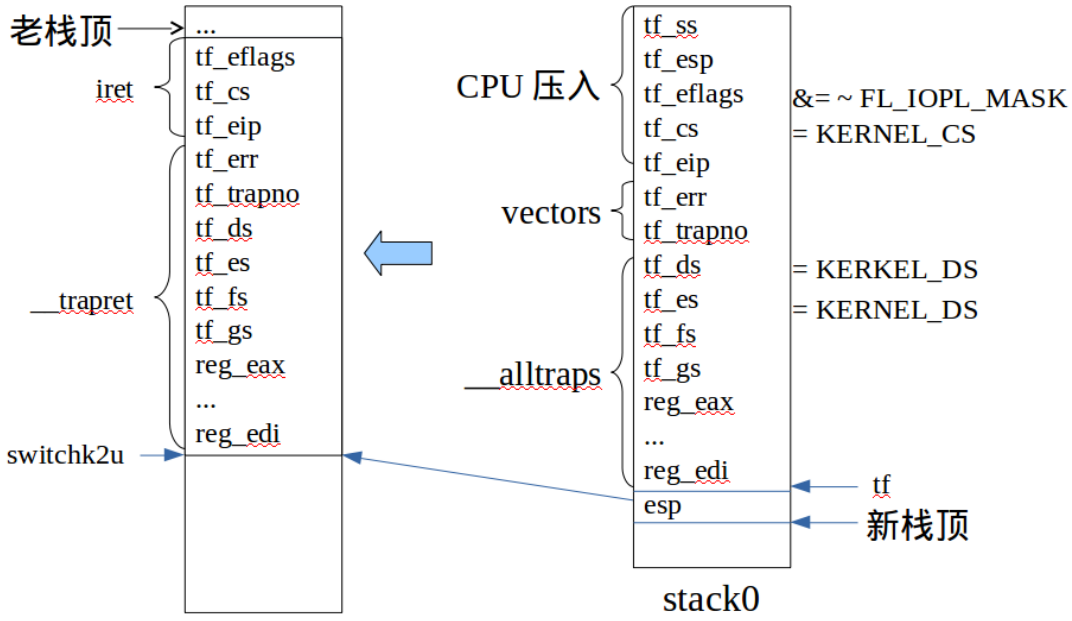

图 2 恢复寄存器的栈情况

[3] https://blog.csdn.net/sinat_30955745/article/details/80976997?utm_source=blogxgwz6

而对于从用户态切换到内核态，是通过在 kern_init 函数中调用 lab1_switch_to_kernel 函数实现的，写出其代码如下：

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

与之前相似，这里使用 int 指令调用了 T_SWITCH_TOK 中断，进而会执行如下代码：

```
    case T_SWITCH_TOK:
        if (tf->tf_cs != KERNEL_CS) {   //如果不在内核态下
            tf->tf_cs = KERNEL_CS;      //内核态代码段
            tf->tf_ds = tf->tf_es = KERNEL_DS;  //内核态数据段
            tf->tf_eflags &= ~FL_IOPL_MASK; //只允许内核的 I/O
            switchu2k = (struct trapframe *)(tf->tf_esp - (sizeof(struct trapframe) - 8));  //设置 switchu2k 指针指向位置
            memmove(switchu2k, tf, sizeof(struct trapframe) - 8);   //复制一份 tf 所指内容
            *((uint32_t *)tf - 1) = (uint32_t)switchu2k;
        }
        break;
```

[4]

[4] https://wenku.baidu.com/view/3d3d0f8587c24028905fc37d.html