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]](#footnote-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:  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  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83 | /\* registers as pushed by pushal \*/  struct pushregs {  uint32\_t reg\_edi;  uint32\_t reg\_esi;  uint32\_t reg\_ebp;  uint32\_t reg\_oesp; /\* Useless \*/  uint32\_t reg\_ebx;  uint32\_t reg\_edx;  uint32\_t reg\_ecx;  uint32\_t reg\_eax;  };  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)); |

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]](#footnote-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：

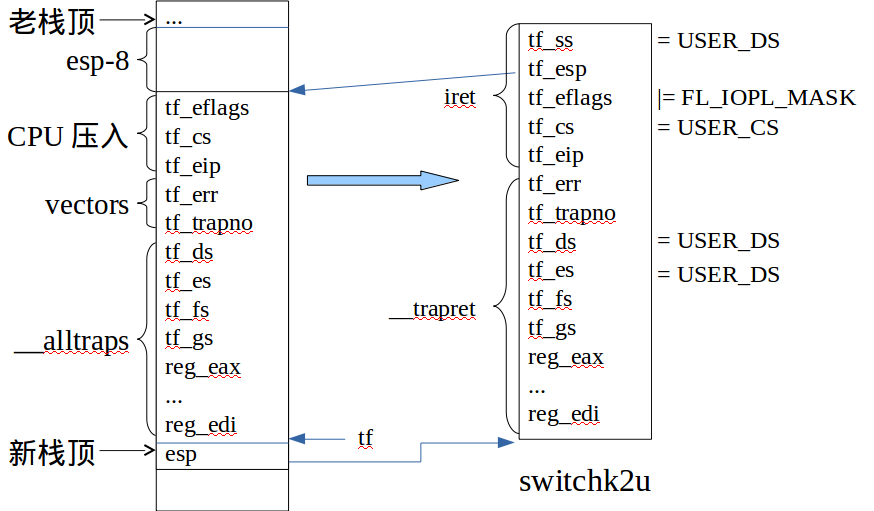


图1 切换到用户态的栈情况

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]](#footnote-3)]

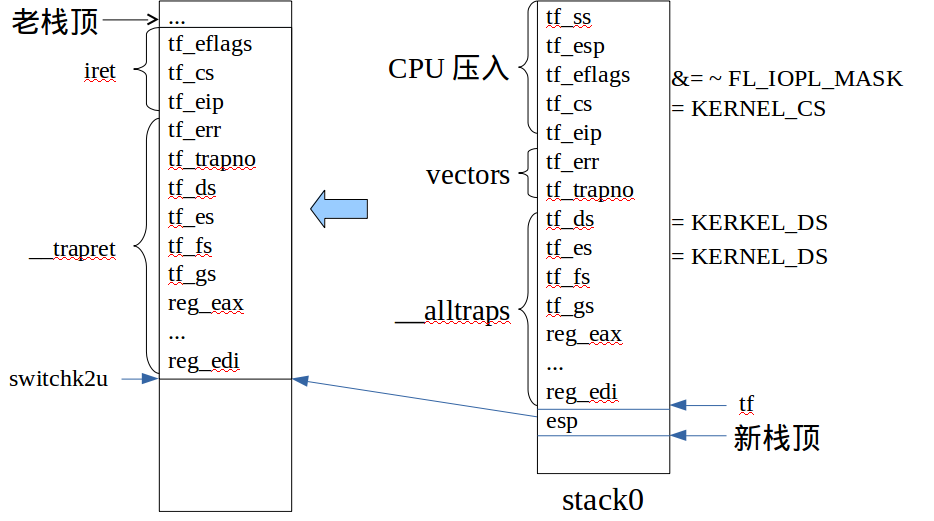


图2 恢复寄存器的栈情况

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]](#footnote-4)]

1. [] https://blog.csdn.net/wdjjwb/article/details/77239612 [↑](#footnote-ref-1)
2. [] https://www.jianshu.com/p/94fec16c5252 [↑](#footnote-ref-2)
3. [] https://blog.csdn.net/sinat\_30955745/article/details/80976997?utm\_source=blogxgwz6 [↑](#footnote-ref-3)
4. [] https://wenku.baidu.com/view/3d3d0f8587c24028905fc37d.html [↑](#footnote-ref-4)