第二次代码阅读报告（Project3）

中断控制的流程分析

1. 首先，系统boot结束之后，进入内核 `main` 函数，其中有一个函数调用 `void tvinit(void);` 作用为初始化中断描述符表；该函数中，循环调用宏定义 `SETGATE` ，将 `vector.S` 中定义的中断向量表载入到 `idt` 表中；这是中断处理的初始化过程；

```
void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}
```

2. 当一个中断发生的时候，首先由硬件处理一部分的保存现场的工作，具体如下：如果处理器此时在用户态下运行，首先需要将用户栈指针 `%esp` 和栈段寄存器 `%ss` 压栈保存，如果处理器此时在内核态下运行则无需上述步骤；接着处理器将当前的标志寄存器 `eflags` ，代码段寄存器 `%cs` ，指令指针寄存器 `%eip` 压栈保存，对于某些特殊的中断，处理器还会将错误信息压栈保存；接着，处理器从中段描述符表中读取当前中断表项的 `%cs` 和 `%eip` 值，从而跳转去 `vector.S` 对应的代码段，将相应的中断号压栈，然后跳转到 `alltrap` 函数；

3. 在 `alltrap` 函数中，处理器将 `%ds` , `%es` , `%fs` , `%gs` 寄存器，以及其他的通用寄存器(`pushal`)压栈保存现场；至此已经构造完成了结构体 `struct trapframe` ，保存着中断发生时的现场；重新设置 `%ds` 和 `%es` 之后，调用 `trap` 函数进入中断控制的C代码部分，该函数的参数已经压栈，只需要将当前栈指针压栈即可；

```
alltraps:
# Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

# Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es

# Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp
```

4. 进入 `trap` 函数之后，针对不同的中断类型，有不同的处理方法：

   1. 系统调用：调用 `syscall()` 函数，从 `trapframe` 结构体中，读取存着中断号的 `%eax` 寄存器；若中断号合法就调用相应的中断函数，并将返回值保存入 `%eax` 寄存器中；若中断号不合法，输出错误信息并将 `%eax` 值设为 `-1` ；

      ```
      //check the trap number to decide whether its a system call
      if(tf->trapno == T_SYSCALL){
        if(myproc()->killed)
            exit();
        myproc()->tf = tf;
        //it's a system call triggered trap; calling syscall()
        syscall();
        if(myproc()->killed)
          exit();
        return;
      }
      ```

   2. 硬件中断：代码中使用 `switch` 结构主要处理了几个硬件中断：时钟、IDE硬盘、键盘、COM1，并且处理不合法的中断；

```
//check if the trap is a hardware interrupts
switch(tf->trapno){
// timer interrupt
case T_IRQ0 + IRQ_TIMER:
if(cpuid() == 0){
  acquire(&tickslock);
  ticks++;
  wakeup(&ticks);
  release(&tickslock);
}
lapiceoi();
break;

// IDE disk interrupt
case T_IRQ0 + IRQ_IDE:
ideintr();
lapiceoi();
break;
case T_IRQ0 + IRQ_IDE+1:
// Bochs generates spurious IDE1 interrupts.
break;

// keyboard interrupt
case T_IRQ0 + IRQ_KBD:
kbdintr();
lapiceoi();
break;

// TODO: figure out what COM1 is
case T_IRQ0 + IRQ_COM1:
uartintr();
lapiceoi();
break;
case T_IRQ0 + 7:
case T_IRQ0 + IRQ_SPURIOUS:
cprintf("cpu%d: spurious interrupt at %x:%x\n",
        cpuid(), tf->cs, tf->eip);
lapiceoi();
break;
}
```

3. exception：首先查看当前进程是否是内核态来判断是内核代码发生错误还是用户代码发生错误；并且根据不同的情况输出一些调试信息；

```
//PAGEBREAK: 13
// not syscall or hardware interrupt; assuming it's a exception
default:
// determine whether it's a kernel error or user error
if(myproc() == 0 || (tf->cs&3) == 0){
  // In kernel, it must be our mistake.
  cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
          tf->trapno, cpuid(), tf->eip, rcr2());
  panic("trap");
}
// In user space, assume process misbehaved.
cprintf("pid %d %s: trap %d err %d on cpu %d "
        "eip 0x%x addr 0x%x--kill proc\n",
        myproc()->pid, myproc()->name, tf->trapno,
        tf->err, cpuid(), tf->eip, rcr2());
myproc()->killed = 1;
```

5. 中断处理的C代码部分结束后，函数返回到 `trapasm.S` 的 `trapret` 入口，至此中断的主要工作已经完成，下面所需要做的就是恢复现场：首先使用 `popal` 命令，弹出通用寄存器，使用 `popl` 命令依次弹出 `%gs` , `%fs` , `%es` , `%ds` 寄存器，恢复栈指针的值，然后调用 `iret` 回到中断发生前的状态；以上就是一个完整的中断控制的流程。

```
# Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

简答题目

1. 问题：分析xv6中IDT（中断描述表）在 `trap` 处理中的作用?

   答： `trap` 发生后，处理器完成一部分保存现场的工作之后，需要找到处理当前中断的代码段，此时就会查询 `IDT` ，找到对应中断向量代码(`vector.S`)的代码段以及偏移，从而将中断信息压栈并调用 `alltrap` 进入中断处理的主要流程；

2. 问题： `trap` 处理前要保存好当前状态，并触发 `handler` ，简要分析该过程?

   答：从 `trapframe` 的结构体代码可以较为清晰的看到保存当前状态的压栈顺序，下面分析详细过程； `trap` 的触发使用 `int` 指令，该指令触发后，处理器会先做一部分的保存现场的工作：如果处于

用户态，就需要将用户的 `%esp` , `%ss` 压栈，如果处于内核态就无需上述步骤，接着将 `eflags` , `%cs` , `%eip` 以及可能的错误信息(uint err)压栈；这就完成了中断发生后处理器保存一部分现场的工作，接下来查询 `IDT` 找到中断向量表的相应代码位置，跳转到 `vector.S` 中的相应位置，这就完成了 `int` 指令，下面开始处理中断的主要工作； `vector.S` 相应代码中主要完成两个任务，第一将中断号压栈(`trapno`)，第二跳转到 `alltrap` 函数；在 `alltrap` 函数中，继续完成保存现场的工作，包括将 `%ds` , `%es` , `%fs` , `%gs` 寄存器，以及其他的通用寄存器(`pushal`)压栈保存，至此已经完成了保存当前状态的工作；将栈指针压栈是为了向 `trap` C函数传参，调用 `trap` 函数就触发了 `handler` ；

3. 问题：分析在 `syscall()` 发生前后栈空间及寄存器的变化，由此简单描述一次系统调用的过程?

   答： `syscall()` 发生前，中断处理程序已经完成了保存现场的工作，栈空间中保存着由 `struct trapframe` 定义的结构体；其中较为有用的是保存在栈内的 `%eax` 寄存器的值，它保存了系统调用的号码数，通过读取该数字，就可以调用相应的 `syscalls[num]()` 函数，执行对应的系统调用代码；以 `sys_exec()` (系统调用编号9)为例，调用该函数后，主要是进行参数的检查，通过 `argstr` , `argint` 等函数检查参数的合法性，并获取函数调用入口，通过 `fetchint` 等函数获取被调用的函数的参数，最后调用 `exec()` 函数调用参数指定的函数；当然如果是系统第一个调用，正常情况下就不会返回了；如果是普通的系统调用，返回 `syscall()` 函数后，将返回值保存入栈中保存的 `trapframe` 结构体中的 `%eax` 项； `syscall()` 函数返回后， `trap()` 函数也返回，回到 `trapret` 的入口，此时将栈中保存的一系列寄存器的值(参见 `trapframe` 定义)，以与压栈顺序相反的顺序弹栈，恢复所有寄存器的值；

阅后心得

1. 对中断流程代码的阅读，让我对于IDT,APIC等概念有了更深的理解，对我完成大作业中网卡硬件中断处理部分有很大的帮助；
2. 其实这段代码我前后阅读了两遍，第一遍阅读完之后还是有点似懂非懂，因为文档中略过了较多的细节，比如 `int` 指令；于是我去简单学习了汇编语言，再次看代码的时候，由于对于底层细节有了一定的了解，对中断处理流程有了新的更深的认识；
3. 对于汇编中部分语句还是有点没能完全理解，比如在将所有寄存器压栈保存之后，重新设置了 `%ds` , `%es` 寄存器，这是我不太明白的地方；以及函数调用过程中栈指针的变化并没有完全理解；对于这些疑惑之处，我认为在通读所有章节后应该能够有一个较为全面的认识。