**最后修改: 2022-3-15**

# 1. 实验要求

本次实验内容包括三方面：

1. 磁盘加载，即引入内核，bootloader加载kernel，由kernel加载用户程序
2. 开始区分内核态和用户态，完善中断机制
3. 通过实现用户态I/O函数介绍基于中断实现系统调用的全过程

## 1.1. 中断机制

为了给用户态提供服务、保护特权级代码，需要大家实现中断机制，完善IDT、TSS、中断处理程序等必要结构。

## 1.2. 实现系统调用库函数`printf`和对应的处理例程

要求printf拥有完整的格式化输出，包括`%d`，`%x`，`%s`,`%c`四种格式转换输出功能。

## 1.3. 键盘按键的串口回显

这一步是为了输入函数做准备，我们提供了键盘按键信息和库函数，需要大家实现由键盘输入到屏幕的显示。

## 1.4. 实现系统调用库函数`getChar`、`getStr`和对应的处理例程

我们需要大家实现两个基础的输入函数，其中`getChar`函数返回键盘输入的一个字符，`getStr`返回键盘输入的一条字符串，不做额外的格式化输入要求。

实验流程如下

1. 由实模式开启保护模式并跳转到bootloader
2. 由bootloader加载kernel
3. 完善kernel相关的初始化设置
4. 由kernel加载用户程序
5. 实现用户需要的库函数
6. 用户程序调用自定义实现的库函数完成格式化输入输出，通过测试代码

# 2. 开始前的建议

本实验框架代码较为复杂，以下列出一些建议希望能更好地帮助大家完成实验：

1. 建议大家在正式开始写代码前把框架通读一遍，这对层次复杂的稍大型工程任务很重要
2. 建议大家使用git管理代码，并安装TODO tree一类的管理标识软件，它可以帮助你更快地找到和梳理清晰实验任务
3. 本次实验量较大，建议大家早点开始准备
4. 你遇到的问题绝大多数都有其他人遇见过，善用搜索引擎可以帮助你更好地解决问题
5. 找配置环境相同的答案更为靠谱
6. 如果遇到觉得代码逻辑没问题但是qemu运行显示异常的情况可能是配置环境的问题，gcc版本太高似乎对实验有影响
7. 如果遇到了boot block is too large的问题，可以采用index引导实验中所用的objcopy命令

8. 问问题前麻烦大家想清楚自己的疑问到底是什么以及有什么自己的想法和思考，不要笼统地问怎么做和怎么错了。问问题建议在qq群中统一提问，这样有助于大家共同学习
9. 祝大家实验顺利！

# 3. 相关资料

## 3.1. 写磁盘扇区

以下代码用于写一个磁盘扇区，框架代码已提供

```
static inline void outLong(uint16_t port, uint32_t data) {
asm volatile("out %0, %1" : : "a"(data), "d"(port));
}

void writeSect(void *src, int offset) {
    int i;
    waitDisk();
    outByte(0x1F2, 1 );
    outByte(0x1F3, offset);
    outByte(0x1F4, offset >> 8 );
    outByte(0x1F5, offset >> 16 );
    outByte(0x1F6, (offset >> 24 ) | 0xE0);
    outByte(0x1F7, 0x30);
    waitDisk();
    for (i = 0 ; i < SECTOR_SIZE / 4 ; i ++) {
        outLong(0x1F0, ((uint32_t *)src)[i]);
    }
}
```

## 3.2. 串口输出

在以往的各种编程作业中，我们都能通过`printf`在屏幕上输出程序的内部状态用于调试，现在做操作系统实验，还没有实现printf等屏幕输出函数，如何通过类似手段进行调试呢？不要慌，在lab2的Makefile我们发现这样的内容：

```
play: os.img
    $(QEMU) -serial stdio os.img
```

其中`-serial stdio`表示将qemu模拟的串口数据即时输出到stdio（即宿主机的标准输出）

在`lab2/kernel/main.c`的第一行就是初始化串口设备`initSerial()`，在之后的代码中，我们就可以通过调用`putChar`（定义在`lab2/kernel/include/device/serial.h`，实现在lab2/kernel/kernel/serial.c）等串口输出函数进行调试或输出log，同时在框架代码中基于`putChar`提供了一个调试接口`assert`（定义在`lab2/kernel/include/common/assert.h`）

有兴趣的同学可以对`putChar`进行封装，实现一个类似`printf`的串口格式化输出`sprintf`

## 3.3 从系统启动到用户程序

我们首先按 OS 的启动顺序来确认一下:

1. 从实模式进入保护模式（lab1）
2. 加载内核到内存某地址并跳转运行（lab1）
3. 初始化串口输出

4. 初始化中断向量表（`initIdt`）
5. 初始化8259a中断控制器（`initIntr`）
6. 初始化 GDT 表、配置 TSS 段（`initSeg`）
7. 初始化VGA设备（`initVga`）
8. 配置好键盘映射表（`initKeyTable`）
9. 从磁盘加载用户程序到内存相应地址（`loadUMain`）
10. 进入用户空间（`enterUserSpace`）
11. 调用库函数 `printf`，`getChar`，`getStr`

内核程序和用户程序将分别运行在内核态以及用户态, 在 Lab1 中我们提到过保护模式除了寻址长度达到 32 位之外, 还能让内核有效地进行权限控制, 在实验的最后, 用户程序擅自修改显存是不被允许的.

**特权级代码的保护**：

- x86 平台 CPU 有 0、1、2、3 四个特权级，其中 level0 是最高特权级，可以执行所有指令
- level3 是最低特权级，只能执行算数逻辑指令，很多特殊操作(例如 CPU 模式转换，I/O 操作指令)都不能在这个级别下进行
- 现代操作系统往往只使用到 level0 和 level3 两个特权级，操作系统内核运行时，系统处于 level0(即 CS寄存器的低两位为 `00b`)，而用戶程序运行时系统处于 level3(即 CS 寄存器的低两位为 `11b`)
- x86 平台使用 CPL、DPL、RPL 来对代码、数据的访存进行特权级检测
    - CPL(current privilege level)为 CS 寄存器的低两位，表示当前指令的特权级
    - DPL(discriptor privilege level)为描述符中的 DPL 字段，表示访存该内存段的最低特权级(有时表示访存该段的最高特权级，比如 Conforming-Code Segment)
    - RPL(requested privilege level)为 DS、ES、FS、GS、SS寄存器的低两位，用于对 CPL 表示的特权级进行补充
    - 一般情况下，同时满足 CPL ≤ DPL，RPL ≤ DPL 才能实现对内存段的访存，否则产生 #GP 异常基于中断机制可以实现对特权级代码的保护

## 3.3.1. 初始化中断向量表

保护模式下 80386 执行指令过程中产生的异常如下表总结

| 向量号 | 助记符 | 描述 | 类型 | 有无出错码 | 源 |
| --- | --- | --- | --- | --- | --- |
| 0 | #DE | 除法错 | Fault | 无 | DIV 和 IDIV 指令 |
| 1 | #DB | 调试异常 | Fault/Trap | 无 | 任何代码和数据的访问 |
| 2 | -- | 非屏蔽中断 | Interrupt | 无 | 非屏蔽外部中断 |
| 3 | #BP | 调试断点 | Trap | 无 | 指令 INT 3 |
| 4 | #OF | 溢出 | Trap | 无 | 指令 INTO |
| 5 | #BR | 越界 | Fault | 无 | 指令 BOUND |
| 6 | #UD | 无效(未定义)操作码 | Fault | 无 | 指令 UD2 或者无效指令 |
| 7 | #NM | 设备不可用(无数学协处理器) | Fault | 无 | 浮点指令或 WAIT/FWAIT 指令 |
| 8 | #DF | 双重错误 | Abort | 有(或零) | 所有能产生异常或 NMI 或 INTR 的指令 |
| 9 |  | 协处理器段越界 | Fault | 无 | 浮点指令(386之后的 IA32 处理器不再产生此种异常) |
| 10 | #TS | 无效TSS | Fault | 有 | 任务切换或访问 TSS 时 |
| 11 | #NP | 段不存在 | Fault | 有 | 加载段寄存器或访问系统段时 |
| 12 | #SS | 堆栈段错误 | Fault | 有 | 堆栈操作或加载 SS 时 |
| 13 | #GP | 常规保护错误 | Fault | 有 | 内存或其他保护检验 |
| 14 | #PF | 页错误 | Fault | 有 | 内存访问 |
| 15 | -- | Intel 保留, 未使用 |  |  |  |
| 16 | #MF | x87FPU浮点错(数字错) | Fault | 无 | x87FPU 浮点指令或 WAIT/FWAIT 指令 |
| 17 | #AC | 对齐检验 | Fault | 有(ZERO) | 内存中的数据访问(486开始) |
| 18 | #MC | Machine Check | Abort | 无 | 错误码(如果有的话)和源依赖于具体模式(奔腾 CPU 开始支持) |
| 19 | #XF | SIMD浮点异常 | Fault | 无 | SSE 和 SSE2浮点指令(奔腾 III 开始) |
| 20-31 | -- | Intel 保留, 未使用 |  |  |  |
| 32-255 | -- | 用户定义中断 | Interrupt |  | 外部中断或 int n 指令 |

以上所列的异常中包括 Fault/Trap/Abort 三种, 当然你也可以称之为错误, 陷阱和终止

- Fault : 一种可被更正的异常, 一旦被更正, 程序可以不失连续性地继续执行, 中断程序返回地址为产生 Fault 的指令
- Trap : 发生 Trap 的指令执行之后立刻被报告的异常, 也允许程序不失连续性地继续执行, 但中断程序返回地址是产生 Trap 之后的那条指令
- Abort : Abort 异常不总是精确报告发生异常的位置, 它不允许程序继续执行, 而是用来报告严重错误.

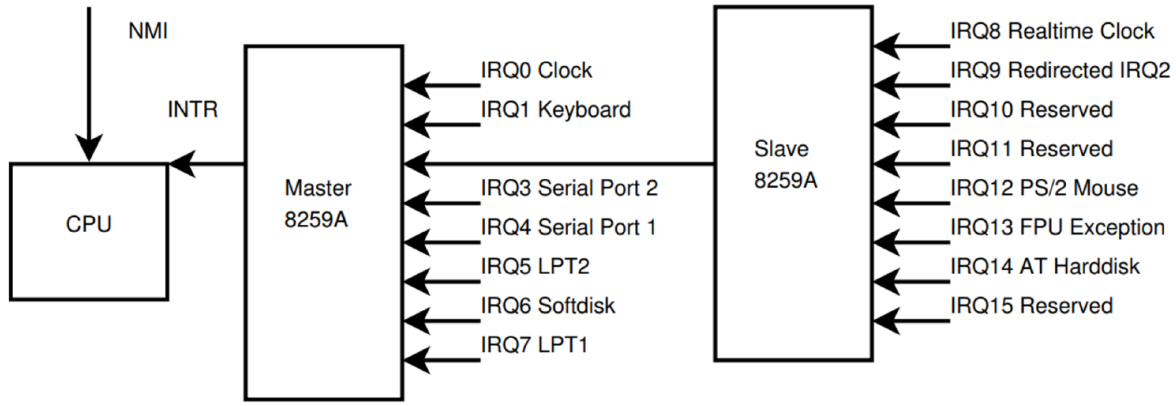
## 3.3.2. 初始化8259a中断控制器

硬件外设I/O：内核的一个主要功能是处理硬件外设的 I/O，CPU 速度一般比硬件外设快很多。多任务系统中，CPU 可以在外设进行准备时处理其他任务，在外设完成准备时处理 I/O；I/O 处理方式包括:轮询、中断、DMA等。

> **Ex1: 计算机系统的中断机制在内核处理硬件外设的 I/O 这一过程中发挥了什么作用?**

中断产生的原因可以分为两种, 一种是外部中断, 即由硬件产生的中断, 另一种就是有指令 int n产生的中断, 下面要讲的是外部中断.

外部中断分别不可屏蔽中断(NMI) 和可屏蔽中断两种, 分别由 CPU 得两根引脚 NMI 和 INTR 来接收, 如图所示




NMI 不可屏蔽, 它与标志寄存器的 IF 没有关系, NMI 的中断向量号为 2 , 在上面的表中已经有所说明（仅有几个特定的事件才能引起非屏蔽中断，例如硬件故障以及或是掉电）. 而可屏蔽中断与 CPU 的关系是通过可编程中断控制器 8259A 建立起来的. 那如何让这些设备发出的中断请求和中断向量对于起来呢? 在 BIOS 初始化 8259A的时候, IRQ0-IRQ7被设置为对应的向量号 `0x08` - `0x0F`, 但是我们发现在保护模式下, 这些向量号已经被占用了, 因此我们不得不重新设置主从8259A(两片级联的8259A).

设置的细节你不需要详细了解, 你只需要知道我们将外部中断重新设置到了 `0x20` - `0x2F` 号中断上

# 3.4. IA-32中断机制

保护模式下的中断源：

- 外部硬件产生的中断(Interrupt)：例如时钟、磁盘、键盘等外部硬件
- CPU 执行指令过程中产生的异常(Exception)：例如除法错( `#DE` )，页错误(#PF)，常规保护错误( `#GP` )
- 由 `int` 等指令产生的软中断(Software Interrupt)：例如系统调用使用的 `int $0x`

前文提到，I/O 设备发出的 IRQ 由 8259A 这个可编程中断控制器(PIC)统一处理，并转化为 8-Bits 中断向量由 INTR 引脚输入 CPU，对于这些这些由8259A控制的可屏蔽中断有两种方式控制：

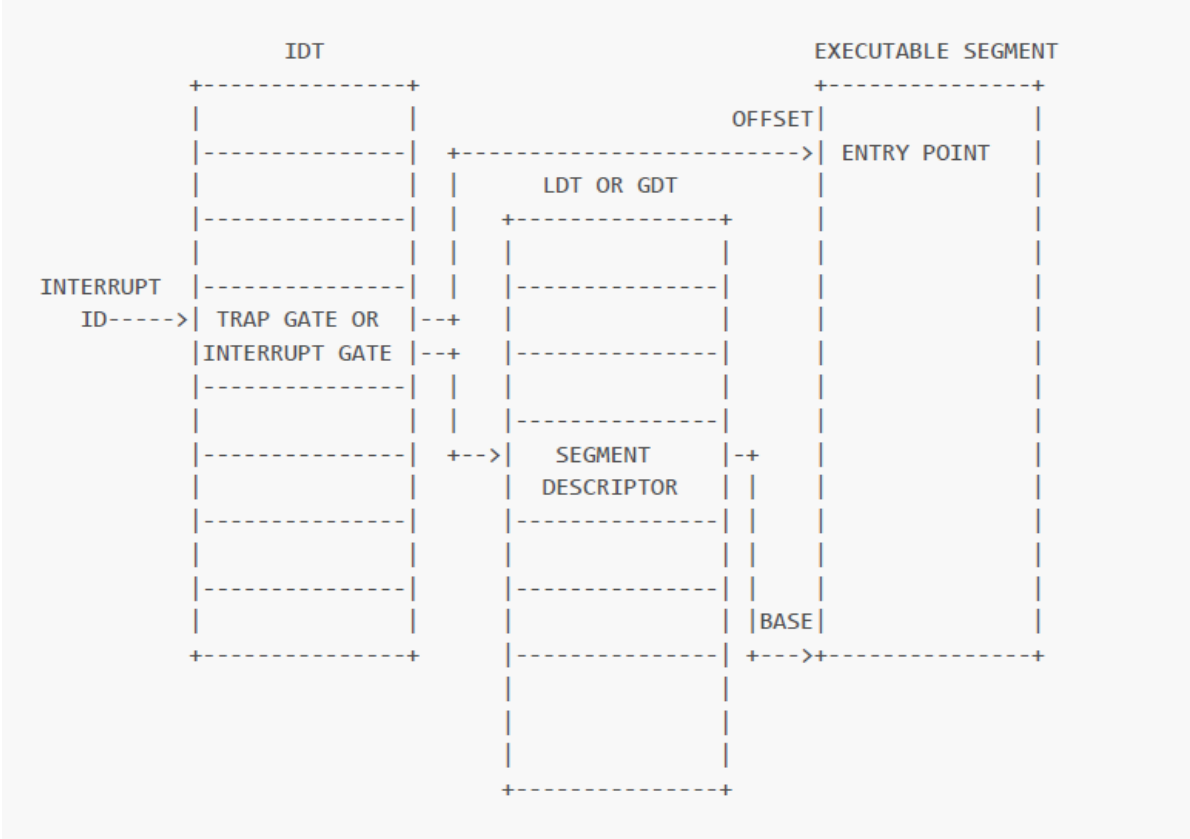
- 通过 `sti` ， `cli` 指令设置 CPU 的 `EFLAGS` 寄存器中的 `IF` 位，可以控制对这些中断进行屏蔽与否
- 通过设置 8259A 芯片，可以对每个 IRQ 分别进行屏蔽

在我们的实验过程中，不涉及对IRQ分别进行屏蔽

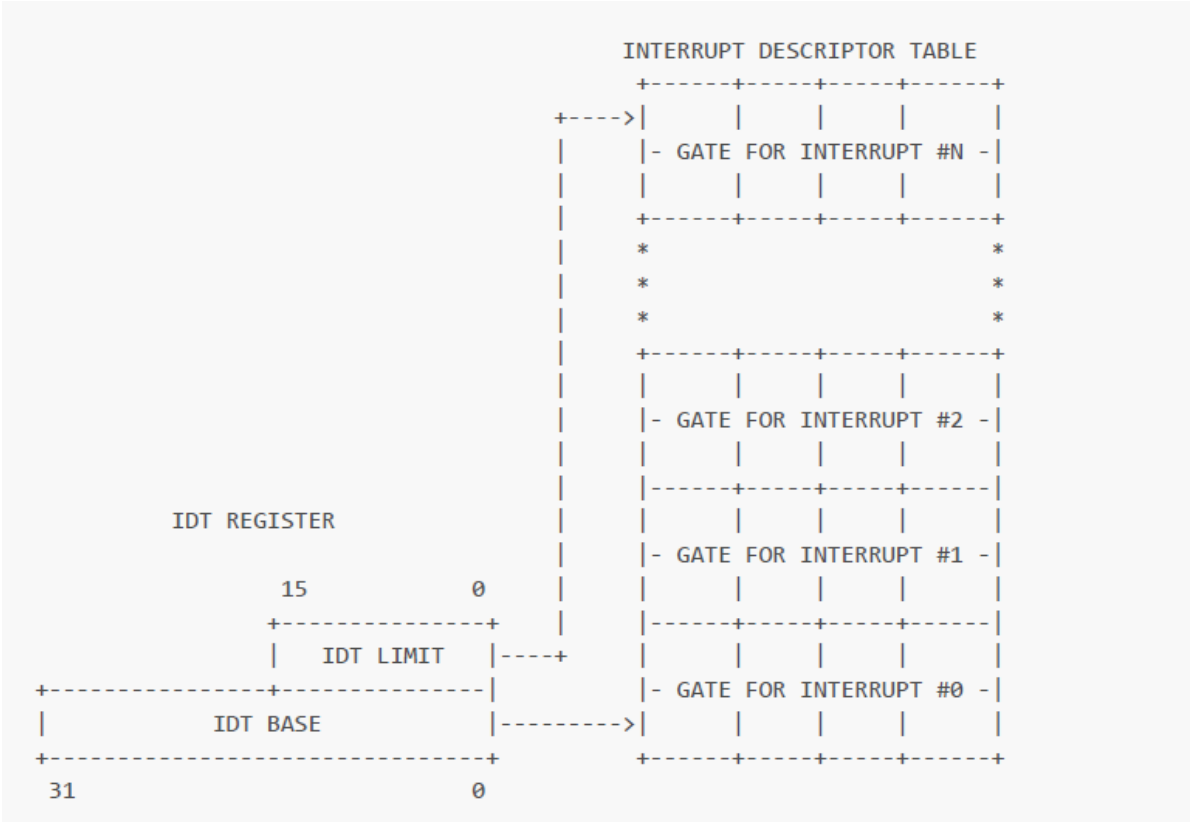
## 3.4.1. IDT

在保护模式下，每个中断(Exception，Interrupt，Software Interrupt)都由一个 8-Bits 的向量来标识，Intel 称其为中断向量，8-Bits表示一共有 256 个中断向量；与 256 个中断向量对应，IDT 中存有 256 个表项，表项称为门描述符(Gate Descriptor)，每个描述符占 8 个字节

中断到来之后，基于中断向量，IA-32硬件利用IDT与GDT这两张表寻找到对应的中断处理程序，并从当前程序跳转执行，下图显示的是基于中断向量寻找中断处理程序的流程



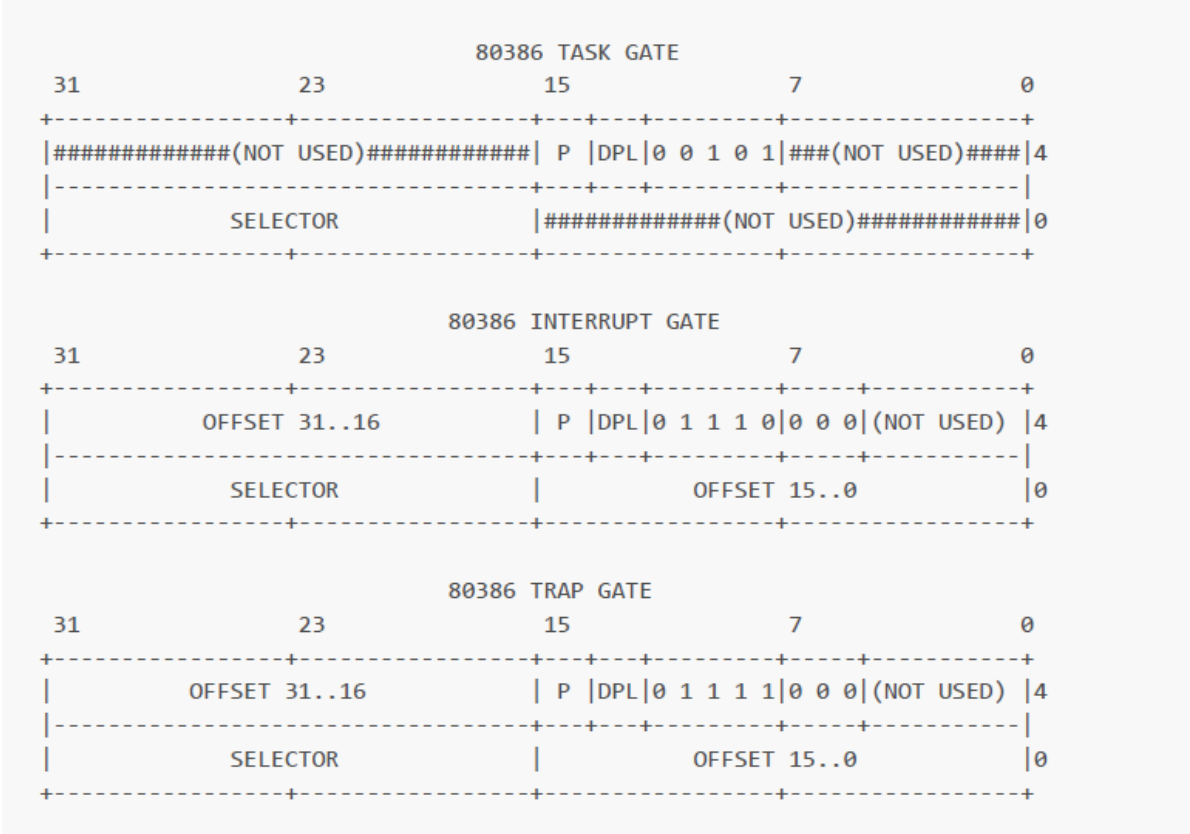

在开启外部硬件中断前，内核需对 IDT 完成初始化，其中IDT的基地址由`IDTR`寄存器（中断描述符表寄存器）保存，可利用`lidt`指令进行加载，其结构如下




IDT中每个表项称为门描述符（Gate Descriptor），门描述符可以分为 3 种

- Task Gate，Intel设计用于任务切换，现代操作系统中一般不使用
- Interrupt Gate，跳转执行该中断对应的处理程序时，`EFLAGS`中的`IF`位会被硬件置为 0，即关中断，以避免嵌套中断的发生
- Trap Gate，跳转执行该中断对应的处理程序时，`EFLAGS`中的`IF`位不会置为 0，也就是说，不关中断

门描述符的结构如下




其中SELECTOR字段表示该中断对应的处理程序所在段的段描述符在GDT中的索引

若中断源为`int`等指令产生的软中断， IA-32硬件处理该中断时还会比较产生该中断的程序的CPL与该中断对应的门描述符的DPL字段，若CPL数值上大于DPL，则会产生General Protect Fault，即#GP异常

## 3.4.2. TSS

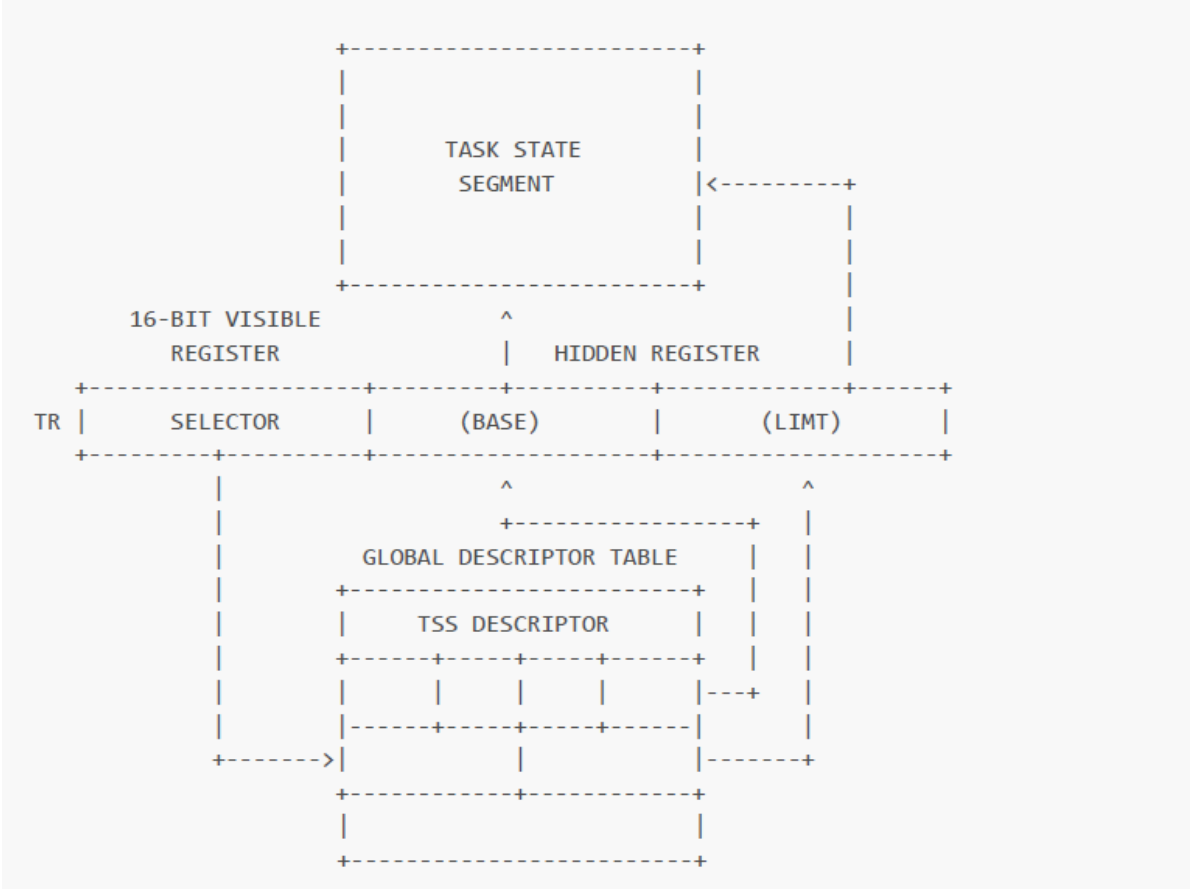
中断会改变程序正常执行的流程，为了方便叙述, 我们称中断到来之前CPU正在执行的工作为A. 中断到来之后,CPU不能再执行A了, 它应该先去处理到来的中断. 因此它应该跳转到一个地方, 去执行中断处理的代码, 结束之后再恢复A的执行. 可以看到, A的执行流程被中断打断了, 为了以后能够完美地恢复到被中断打断时的状态, CPU在处理中断之前应该先把A的状态保存起来, 等到中断处理结束之后, 根据之前保存的信息把计算机恢复到A被打断之前的状态. 这样A就可以继续运行了, 在它看来, 中断就好像没有发生过一样.

接下来的问题是, 哪些内容表征了A的状态? CPU又应该将它们保存到哪里去? 在IA-32中, 首先当是`EIP`(instruction pointer)了, 它指示了A在被打断的时候正在执行的指令; 然后就是`EFLAGS`(各种标志位)和`CS`(代码段, CPL). 由于一些特殊的原因, 这三个寄存器的内容必须由硬件来保存. 此外, 通用寄存器(GPR, general propose register)的值对A来说还是有意义的, 而进行中断处理的时候又难免会使用到寄存器. 但硬件并不负责保存它们, 因此我们还需要手动保存它们的值.

要将这些信息保存到哪里去呢? 一个合适的地方就是程序的堆栈. 中断到来时, 硬件会自动将`EFLAGS`, `CS`, `EIP`三个寄存器的值保存到堆栈上. 此外, IA-32提供了`pusha`/`popa`指令, 用于把通用寄存器的值压入/弹出堆栈, 但你需要注意压入的顺序(请查阅i386手册). 如果希望支持中断嵌套（即在进行优先级低的中断处理的过程中, 响应另一个优先级高的中断 ）， 那么堆栈将是保存信息的唯一选择. 如果选择把信息保存在一个固定的地方, 发生中断嵌套的时候, 第一次中断保存的状态信息将会被优先级高的中断处理过程所覆盖!

IA-32借助`TR`和`TSS`来确定保存`EFLAGS`， `CS`， `EIP`这些寄存器信息的新堆栈

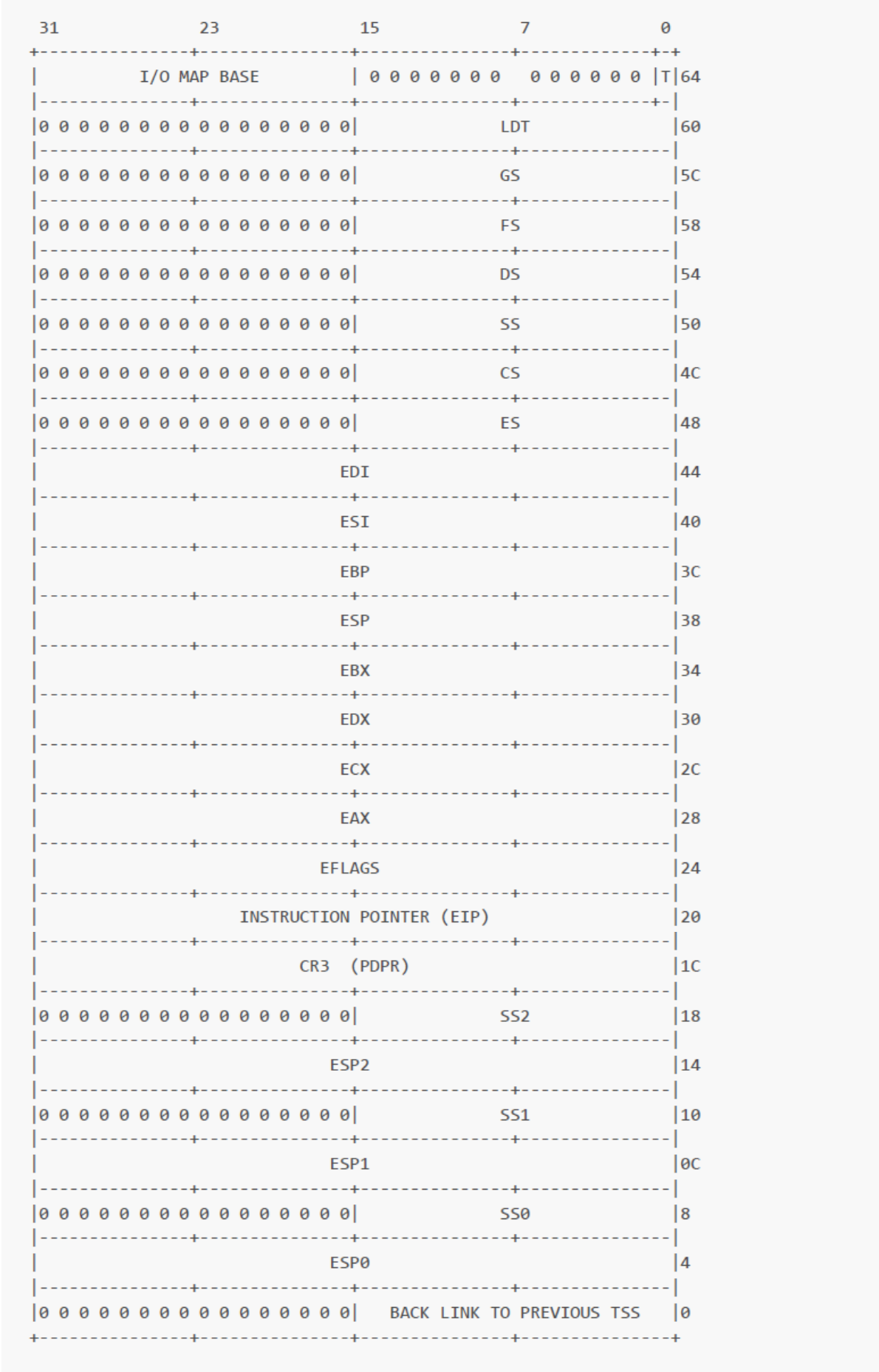
`TR`（Task state segment Register）是 16 位的任务状态段寄存器，结构和`CS`这些段寄存器完全一样，它存放了GDT的一个索引，可以使用`ltr`指令进行加载，通过`TR`可以在GDT中找到一个TSS段描述符，索引过程如下



TSS是任务状态段，不同于代码段、数据段，TSS是一个系统段，用于存放任务的状态信息，主要用在硬件上下文切换

TSS提供了 3 个堆栈位置（`SS`和`ESP`），当发生堆栈切换的时候，CPU将根据目标代码特权级的不同，从TSS中取出相应的堆栈位置信息进行切换，例如我们的中断处理程序位于ring0，因此CPU会从TSS中取出`SS0`和`ESP0`进行切换

为了让硬件在进行堆栈切换的时候可以找到新堆栈，内核需要将新堆栈的位置写入TSS的相应位置，TSS中的其它内容主要在硬件上下文切换中使用，但是因为效率的问题大多数现代操作系统都不使用硬件上下文切换，因此TSS中的大部分内容都不会使用，其结构如下图所示




> **Ex2: IA-32提供了4个特权级, 但TSS中只有3个堆栈位置信息, 分别用于ring0, ring1, ring2的堆栈切换。为什么**
> **TSS中没有ring3的堆栈信息?**

加入硬件堆栈切换之后, 中断到来/从中断返回的硬件行为如下

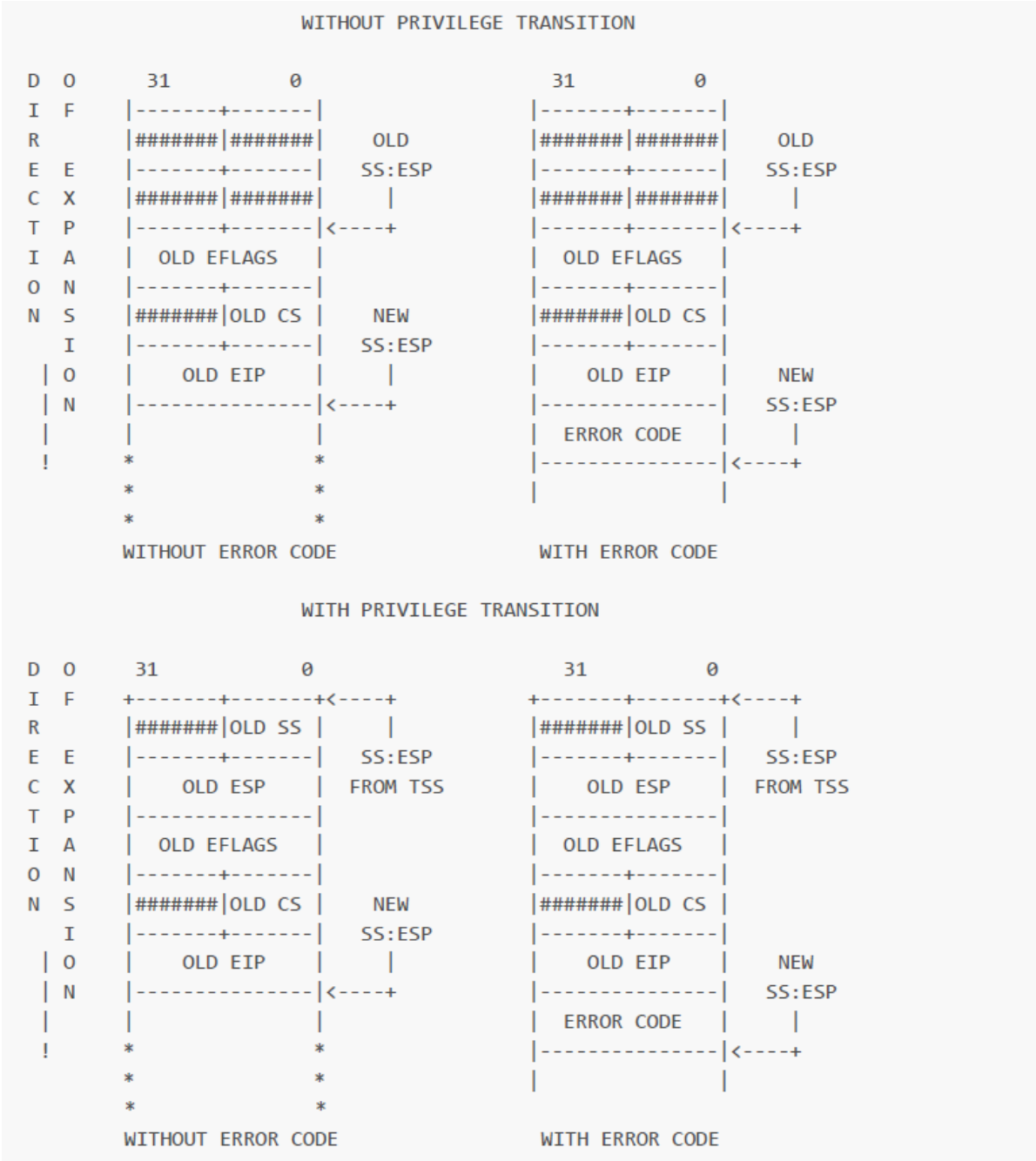
```
old_CS = CS
old_EIP = EIP
old_SS = SS
old_ESP = ESP
target_CS = IDT[vec].selector
target_CPL = GDT[target_CS].DPL
if(target_CPL < GDT[old_CS].DPL)
    TSS_base = GDT[TR].base
    switch(target_CPL)
        case 0:
            SS = TSS_base->SS
            ESP = TSS_base->ESP
        case 1:
            SS = TSS_base->SS
            ESP = TSS_base->ESP
        case 2:
            SS = TSS_base->SS
            ESP = TSS_base->ESP
    push old_SS
    push old_ESP
push EFLAGS
push old_CS
push old_EIP

################### iret ####################

old_CS = CS
pop EIP
pop CS
pop EFLAGS
if(GDT[old_CS].DPL < GDT[CS].DPL)
pop ESP
pop SS
```

硬件堆栈切换只会在目标代码特权级比当前堆栈特权级高的时候发生，即`GDT[target_CS].DPL < GDT[SS].DPL`（这里的小于是数值上的），当`GDT[target_CS].DPL = GDT[SS].DPL`时，CPU将不会进行硬件堆栈切换

下图显示中断到来后内核堆栈的变化




## 3.5. 系统调用

系统调用的入口定义在`lib`下的`syscall.c`，在`syscall`函数里可以使用嵌入式汇编，先将各个参数分别赋值给`EAX，EBX，ECX，EDX，EDX，EDI，ESI`，然后约定将返回值放入`EAX`中（把返回值放入`EAX`的过程是我们需要在内核中实现的），接着使用`int`指令陷入内核，在`lab2/lib/syscall.c`中实现了如下的`syscall`函数：

```
int32_t syscall(int num, uint32_t a1,uint32_t a2,
                uint32_t a3, uint32_t a4, uint32_t a5)
{
    int32_t ret = 0 ;
    uint32_t eax, ecx, edx, ebx, esi, edi;
    asm volatile("movl %%eax, %0":"=m"(eax));
    asm volatile("movl %%ecx, %0":"=m"(ecx));
    asm volatile("movl %%edx, %0":"=m"(edx));
    asm volatile("movl %%ebx, %0":"=m"(ebx));
    asm volatile("movl %%esi, %0":"=m"(esi));
    asm volatile("movl %%edi, %0":"=m"(edi));
    asm volatile("movl %0, %%eax"::"m"(num));
```

```
    asm volatile("movl %0, %%ecx"::"m"(a1));
    asm volatile("movl %0, %%edx"::"m"(a2));
    asm volatile("movl %0, %%ebx"::"m"(a3));
    asm volatile("movl %0, %%esi"::"m"(a4));
    asm volatile("movl %0, %%edi"::"m"(a5));
    asm volatile("int $0x80");
    asm volatile("movl %%eax, %0":"=m"(ret));
    asm volatile("movl %0, %%eax"::"m"(eax));
    asm volatile("movl %0, %%ecx"::"m"(ecx));
    asm volatile("movl %0, %%edx"::"m"(edx));
    asm volatile("movl %0, %%ebx"::"m"(ebx));
    asm volatile("movl %0, %%esi"::"m"(esi));
    asm volatile("movl %0, %%edi"::"m"(edi));
    return ret;
}
```

> **Ex3: 我们在使用eax, ecx, edx, ebx, esi, edi前将寄存器的值保存到了栈中，如果去掉保存和恢复的步骤，从内核返回之后会不会产生不可恢复的错误?**

`int` 指令接收一个8-Bits的立即数为参数，产生一个以该操作数为中断向量的软中断，其流程分为以下几步

1. 查找 `IDTR` 里面的IDT地址，根据这个地址找到IDT，然后根据IDT找到中断向量的门描述符
2. 检查CPL和门描述符的DPL，如果CPL数值上大于DPL，产生#GP异常，否则继续
3. 如果是一个ring3到ring0的陷入操作，则根据 `TR` 寄存器和GDT，找到TSS在内存中的位置，读取其中的 `SS0` 和 `ESP0` 并装载则向堆栈中压入 `SS` 和 `ESP`，注意这个 `SS` 和 `ESP` 是之前用户态的数据
4. 压入 `EFLAGS，CS，EIP`
5. 若门描述符为Interrupt Gate，则修改 `EFLAGS` 的 `IF` 位为 0
6. 对于某些特定的中断向量，压入Error Code
7. 根据IDT表项设置 `CS` 和 `EIP`，也就是跳转到中断处理程序执行

中断处理程序执行结束，需要从ring0返回ring3的用户态的程序时，使用 `iret` 指令

`iret` 指令流程如下

1. `iret` 指令将当前栈顶的数据依次Pop至 `EIP，CS，EFLAGS` 寄存器
2. 若Pop出的 `CS` 寄存器的CPL数值上大于当前的CPL，则继续将当前栈顶的数据依次Pop至 `ESP`，`SS` 寄存器
3. 恢复CPU的执行

**系统调用的参数传递**

每个系统调用至少需要一个参数，即系统调用号，用以确定通过中断陷入内核后，该用哪个函数进行处理；普通 C 语言的函数的参数传递是通过将参数从右向左依次压入堆栈来实现；系统调用涉及到用戶堆栈至内核堆栈的切换，不能像普通函数一样直接使用堆栈传递参数；框架代码使用 `EAX，EBX` 等等这些通用寄存器从用戶态向内核态传递参数：

框架代码 `kernel/irqHandle.c` 中使用了 `TrapFrame` 这一数据结构，其中保存了内核堆栈中存储的 7 个寄存器的值，其中的通用寄存器的取值即是通过上述方法从用戶态传递至内核态，并通过 `pushal` 指令压入内核堆栈

## 3.6. 键盘驱动

以下代码用于获取键盘扫描码，每个键的按下与释放都会分别产生一个键盘中断，并对应不同的扫描码；对于不同类型的键盘，其扫描码也不完全一致

```
uint32_t getKeyCode() {
    uint32_t code = inByte(0x60);
    uint32_t val = inByte(0x61);
    outByte(0x61, val | 0x80);
    outByte(0x61, val);
    return code;
}
```

# 4. 解决思路

磁盘加载不再赘述，关于中断机制，你可以单独完成，也可以结合printf或是按键串口回显的逻辑完成中断。下面以按键回显为例子，来说下按键和中断的思路：

## 4.1. 键盘按键的串口回显

当用户按键（按下或释放）时，键盘接口会得到一个代表该按键的键盘扫描码，同时产生一个中断请求。键盘中断服务程序先从键盘接口取得按键的扫描码，然后根据其扫描码判断用户所按的键并作相应的处理，最后通知中断控制器本次中断结束并实现中断返回。

### 4.1.1. 设置门描述符

要想加上键盘中断的处理，首先要在IDT表中加上键盘中断对应的门描述符，根据前文，8259a将硬件中断映射到键盘中断的向量号`0x20-0x2F`，键盘为IRQ1，所以键盘中断号为`0x21`，框架代码也提供了键盘中断的处理函数`irqKeyboard`，所以需要同学们在`initIdt`中完成门描述符设置。

值得注意的一点是：硬件中断不受DPL影响，8259A的 15 个中断都为内核级可以禁止用户程序用int指令模拟硬件中断

完成这一步后，每次按键，内核会调用`irqKeyboard`进行处理

### 4.1.2. 完善中断服务例程

追踪`irqKeyboard`的执行，最终落到`keyboardHandle`，同学们需要在这里利用 **键盘驱动接口** 和 **串口输出接口**完成键盘按键的串口回显，完成这一步之后，你就能在stdio显示你按的键，另外，你也可以采用我们熟悉的显存的方式，将按键直接打印在屏幕上

## 4.2. 实现`printf`的处理例程

和键盘中断一样，对系统调用来说，同样需要设置门描述符，本次实验将系统调用的中断号设为`0x80`，中断调用的处理函数为`irqSyscall`，DPL设置为用户级；以后所有的系统调用都是通过`0x80`号中断完成，不过通过不同的系统调用号（`syscall`的第一个参数）选择不同的处理例程

在用户调用`int 0x80`之后，最终的写显存相关内容在`syscallPrint`中，需要同学们填充完成

```
void syscallPrint(struct TrapFrame *tf) {
    int sel = //TODO: segment selector for user data, need further modification
    char *str = (char*)tf->edx;
    int size = tf->ebx;
    int i = 0 ;
    int pos = 0 ;
    char character = 0 ;
    uint16_t data = 0 ;
    asm volatile("movw %0, %%es"::"m"(sel));
    for (i = 0 ; i < size; i++) {
        asm volatile("movb %%es:(%1), %0":"=r"(character):"r"(str+i));
```

```
        // TODO: 完成光标的维护和打印到显存
    }
    updateCursor(displayRow, displayCol);
}
```

**提示**

`asm volatile("movb %%es:(%1), %0":"=r"(character):"r"(str+i));`表示将待print的字符串`str`的第`i`个字符赋值给`character`

以下这段代码可以将字符`character`显示在屏幕的`displayRow`行`displayCol`列

```
data = character | (0x0c << 8 );
pos = ( 80 *displayRow+displayCol)* 2 ;
asm volatile("movw %0, (%1)"::"r"(data),"r"(pos+0xb8000));
```

需要注意的是碰上`\n`以及换行，滚屏的处理，QEMU模拟的屏幕的大小是`80*25`

完成这一步后，用户调用`printf`就能在屏幕上进行输出了

## 4.3. 完善`printf`的格式化输出

在框架代码中已经提供了`printf`最基本的功能

```
void printf(const char *format,...){
    int i= 0 ; // format index
    char buffer[MAX_BUFFER_SIZE];
    int count= 0 ; // buffer index
    int index= 0 ; // parameter index
    void *paraList=(void*)&format; // address of format in stack
    int state= 0 ; // 0: legal character; 1: '%'; 2: illegal format
    int decimal= 0 ;
    uint32_t hexadecimal= 0 ;
    char *string= 0 ;
    char character= 0 ;
    while(format[i]!= 0 ){
        buffer[count]=format[i];
        count++;
        //TODO in lab2
    }
    if(count!= 0 )
    syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)count, 0 , 0 );
}
```

> **Ex4: 查阅相关资料，简要说明一下`%d`，`%x`，`%s`，`%c`四种格式转换说明符的含义。**

## 4.4. 实现`getChar`，`getStr`的处理例程

这两个函数的处理方式比较灵活，`getChar`相对来说要容易一些，你可以等按键输入完成的时候，将末尾字符通过eax寄存器传递回来。需要注意的是，在用户态向内核态传递参数的过程中，eax既扮演了参数的角色，又扮演了返回值的角色。这一段在汇编代码在`lab2/lib/syscall.c`中有体现

```
asm volatile("int $0x80");
asm volatile("movl %%eax, %0":"=m"(ret));
```

`getStr`的实现要更为麻烦一些，涉及到字符串的传递。我们要知道的是，即使采用了通用寄存器做为中间桥梁，字符串地址作为参数仍然涉及到了不同的数据段。实际上，在处理`printf`的时候我们就遇见过这个问题，它的解决方式是，在内核态进行了段描述子的切换**内核态可以访问用户态的数据段，但是用户态不可以越级访问内核态的数据。**这就要求我们不能采用把内核获取的显存字符串地址传出来，而得是在内核态提前开辟一块字符串地址，将这个地址传进内核态。

当然，`getStr`也可以不局限于这一种实现思路，能奏效的实现方式都被接受。

## 4.5. 测试用例

最后，我们还为大家准备了用户态I/O调用的测试代码，放置在用户程序主函数中，大家在实现功能时可以把这些复杂的测试用例先注释掉，自己写一些简单的调用进行验证。

然后在你觉得大功告成了之后，将上述测试用例取消注释，如果你能完全通过测试，恭喜你，lab2已经全部完成了，下次再见！

# 5. 代码框架

```
lab2-STUID #自行修改后打包(.zip)提交
├── lab
| ├── Makefile
| ├── app #用户代码
| | ├── Makefile
| | └── main.c #主函数
| ├── bootloader #引导程序
| | ├── Makefile
| | ├── boot.c
| | ├── boot.h
| | └── start.S
| ├── kernel
| | ├── Makefile
| | ├── include #头文件
| | | ├── common
| | | | ├── assert.h
| | | | ├── const.h
| | | | └── types.h
| | | ├── common.h
| | | ├── device
| | | | ├── disk.h
| | | | ├── keyboard.h
| | | | ├── serial.h
| | | | └── vga.h
| | | ├── device.h
| | | ├── x
| | | | ├── cpu.h
| | | | ├── io.h
| | | | ├── irq.h
| | | | └── memory.h
| | | └── x86.h
| | ├── kernel #内核代码
| | | ├── disk.c #磁盘读写API
| | | ├── doIrq.S #中断处理
| | | ├── i8259.c #重设主从8259A
| | | ├── idt.c #初始化中断描述
| | | ├── irqHandle.c #中断处理函数
| | | ├── keyboard.c #初始化键码表
```

```
| | | ├── kvm.c #初始化 GDT 和加载用户程序
| | | ├── serial.c #初始化串口输出
| | | └── vga.c
| | ├── lib
| | | └── abort.c
| | └── main.c #主函数
| ├── lib #库函数
| | ├── lib.h
| | ├── syscall.c #系统调用入口
| | └── types.h
| └── utils
| ├── genBoot.pl #生成引导程序
| └── genKernel.pl #生成内核程序
└── report
└── stuid.pdf
```

# 6. 相关资源

US-QWERTY键盘扫描码

# 7. 作业提交

本次作业需提交可通过编译的实验相关源码与报告，提交前请确认`make clean`过，报告中要包含四道思考题的答案以及关键代码的实现方法。
提交的最后结果应该要能完整实现三个系统调用库函数`printf`，`getChar`，`getStr`。