> 最后修改: 2020-3-7

1. 实验要求

本次实验内容包括两部分：

1. 编写一个格式化程序，按照文件系统的数据结构，构建磁盘文件（即`os.img`）
2. 开始区分内核态和用户态；通过实现键盘中断了解硬件中断；通过实现用户态`printf`介绍基于中断实现系统调用的全过程

1.1. 格式化程序

带文件系统的磁盘构建

- Bootloader占os.img的0号扇区
- Kernel占os.img的1-201号扇区
- os.img剩下的空间格式化为文件系统

Bootloader和Kernel不包含在文件系统内，lab2主要涉及文件系统部分的格式化

1. 文件系统部分包含8192个扇区，每个扇区512字节，共4MB
2. 将这4MB空间格式化为简化MINIX文件系统
3. 建立文件系统的目录
4. 将用户程序写入到文件系统中

1.2. 键盘按键的串口回显

当用户按键，会发出一个键盘中断到内核，内核调用键盘中断的处理函数实现串口回显

1.3. 实现系统调用库函数`printf`和对应的处理例程

实验流程如下

1. 用户程序调用自定义实现的库函数`printf`完成格式化输出
2. `printf`基于中断陷入内核，由内核完成在vga映射的显存地址中写入内容，完成字符串的打印

1.4. 完善`printf`的格式化输出

格式化输入输出的测试用例在`lab2/app/main.c`中已给出

2. 相关资料

2.1. 层次结构文件系统

在lab1中，我们采用`readSec`直接进行磁盘指定扇区的读写，约定app.bin存放在磁盘的第1个扇区后，因此只要把约定好的内容读进来就行。但是操作系统需要和很多文件打交道，我们不可能强行约定所有的文件位置，同时如果要对文件进行添加、修改、删除等操作，原有的约定就会打破，我们需要文件系统来管理文件。

一个完整的文件系统不外乎包含以下功能:

- 按名存取
- 目录的建立和维护
- 逻辑文件到物理文件的转换
- 存储空间的分配和管理
- ~~数据保密,保护和共享~~
- 提供一组用户使用的操作

实验2的文件系统只涉及到文件系统的格式化、目录的建立和文件的写入。

2.1.1. 文件名与iNode

使用字符串作为文件名更符合我们"见名知意"的使用习惯. 需要考虑的问题是文件名应该存放在哪里, 我们有如下两种选择:

- 随文件内容一起存放
- 另起一片区域存放

文件名随文件内容一起存放会产生一个问题, 应用程序读写一个文件的时候, 需要跳过文件名等信息, 稍有不慎就会覆盖了这些信息, 给文件操作造成不便. 因此目前大多数文件系统都采用第二种方式存放文件名. minix和ext文件系统使用iNode这种数据结构来记录文件名等信息, 并在磁盘中设有一片连续区域来存放iNode信息.

既然文件名不随文件内容一起存放, 如何通过文件名找到相应的文件内容呢? 我们需要将文件的存储位置信息也记录到iNode结构中:

```
struct Inode {
  char filename[32];
  off_t offset;
};
```

2.1.2. 分配与组织

根据文件大小动态分配空间是文件系统务必实现的一个重要需求. 由于文件大小不固定, 我们需要在iNode结构中加入表示文件大小的属性:

```
struct Inode {
  char filename[32];
  size_t size;
  off_t offset;
};
```

顺序存放的特性十分容易实现, 但使用起来却容易产生很多问题, 一个很大的问题就是容易产生磁盘碎片: 当一些很小的文件被删除时, 它们所占的磁盘空间很难被再次利用, 因为这些磁盘空间连比这些小文件大一点的文件都放不下.

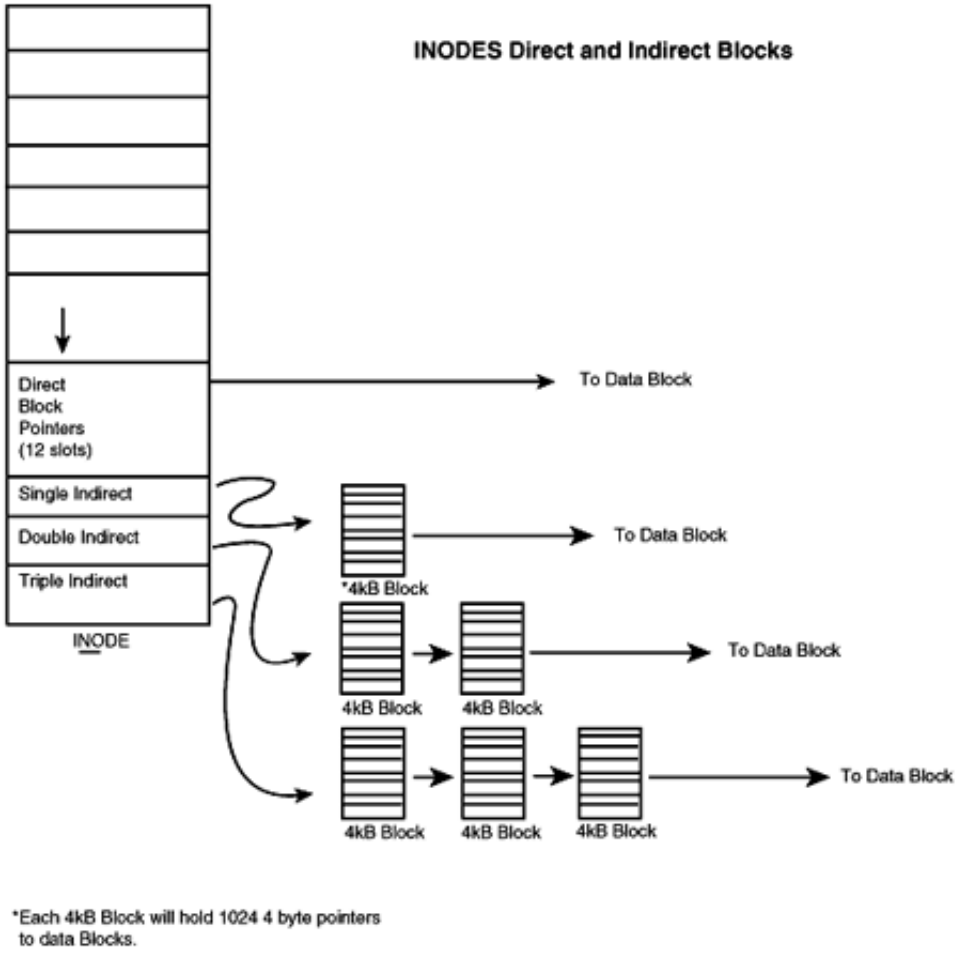
我们需要一种策略将这些碎片组织起来, 以方便再次使用它们. 我们引入"分配最小单元"的概念, 在文件系统的范畴中, 这个概念叫做"块"(block). 每一个文件由若干个块组成, 这些块不需要顺序存放, 只要按照某种方法组织起来. 这样, 即使小文件被删除, 他们所占用的块也可以得到有效的利用. 但这样一来我们又遇到了新的问题.

如何将多个块组织起来

块的组织方式对文件系统的性能有很大的影响, 我们需要考虑以下方面的问题:

- 需要支持文件的随机访问(访问一个文件的不同部分的效率都应该一样), 链表形式的组织方式显然不太合适, 因为访问的位置越靠后, 所花时间越长
- 需要方便地支持块的插入和删除, 这是因为文件的大小经常发生变化, 数组形式的组织方式显然不太合适, 因为向数组中插入元素可能需要申请更大的空间, 涉及到整个数组的拷贝, 频繁地进行这样的操作会严重影响效率, 磁盘的寿命也会有所下降
- 组织方式带来的额外开销不能太大, 一种需要花费大量磁盘空间来维护的组织方式显然是不被接受的

ext文件系统采用如下方式来将不同的块组织起来:



相应的iNode结构为:

```
typedef uint32_t block_t;
struct Inode {
  char filename[32];
  size_t size;
  block_t index[15];
};
```

其中:

- index[0]~index[11]为数据块的索引, 它们指向的块中存储了文件的数据;
- index[12]为一级索引块的索引, 它指向的块并没有直接存储文件的数据, 而是存储了若干数据块的索引. 若使用数组的表示方式, 则index[12][0], index[12][1]...才是数据块的索引, 它们指向的块中存储了文件的数据;
- index[13]为二级索引块的索引, 它指向的块存储了若干一级索引块的索引;
- index[14]为三级索引块的索引(上图中没有画出), 它指向的块存储了若干二级索引块的索引.

这种数据结构使得文件中任意数据块的访问, 插入和删除操作的时间复杂度都是O(1), 因为访问文件中任何位置的数据最多在访问5个磁盘块后达到; 另一方面, 这种数据结构的开销取决于索引块的数目, 小文件只需要用到前几个index就足够了, 不需要使用索引块; 而对于大型文件, 在块大小为4KB, 块索引号为32位整数的情况下, 用到的索引块的大小大约为文件大小的千分之一.

在实验代码中会对索引过程稍作简化

> ### 与存储管理机制进行比较
>
> 分别考虑内存分段机制, ext文件系统和内存分页机制, 它们之间是否有某种联系? 回忆i386分页机制, 为什么ext文件系统没有采用类似于分页机制中对物理页的组织方式, 通过固定的两级索引来寻找物理页?

块应该取多大

根据前文的分析, 块不应该太大, 否则将会严重影响文件系统的存储效率; 而太小的块将会使得需要使用大量的块来存储相同大小的文件, 大大增加了块组织的额外开销. 下表列出了块大小不同对文件系统一些参数的影响, 当然, 这些都是理论值, 实际上一般不会使用这么大的文件.

块大小	1KB	2KB	4KB
一个文件最大大小	16GB	256GB	2TB
文件系统最大大小	4TB	8TB	16TB

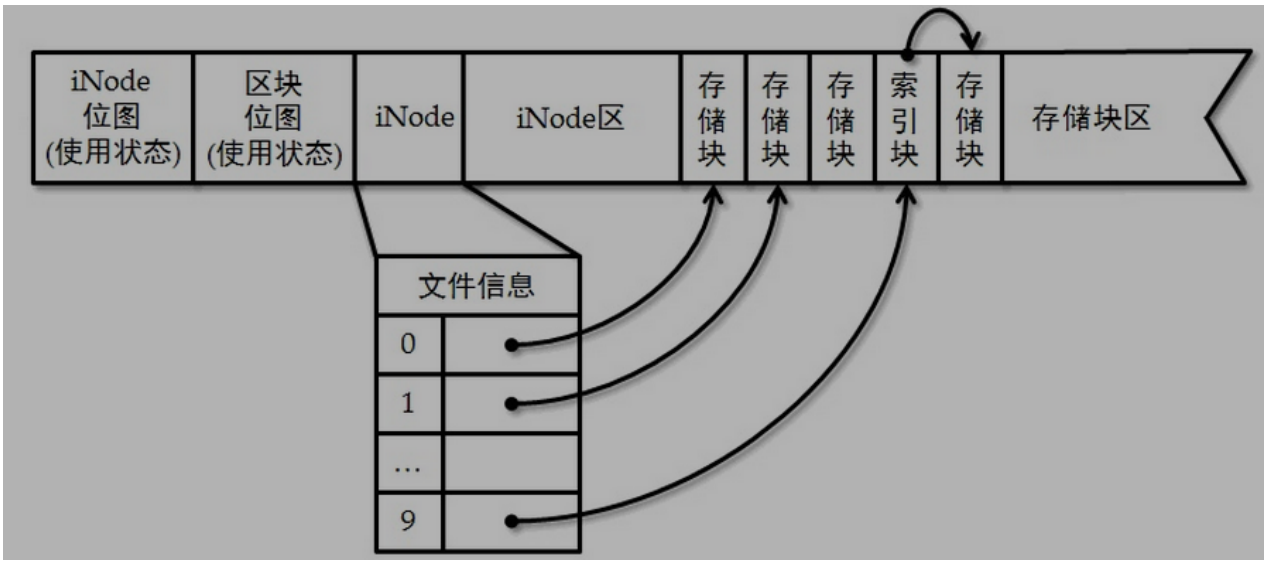
> ### 表上参数的计算
>
> 你知道上表中的数据是如何计算出来的吗?

> ### "幽灵"空间
>
> 在磁盘上创建一个新文本文件, 往里面写入一个字符后, 查看该文件的大小:

> - 在Windows中, 有一项叫Size on disk的数据
> - 在Linux中, 使用`ls -ls`命令, 输出的第一列即为文件占用的磁盘空间(单位:KB)
>
> 这些显示的数值远远大于一个字节, 你知道为什么吗?

到此, 一个文件系统的框架已经成型了, 磁盘(RAMDISK)被划分成四个区域:

- iNode位图区, 用于记录iNode结构的使用状态
- 区块位图区, 用于记录磁盘块的使用状态
- iNode区, 用于存放iNode结构, 可以看做iNode结构的数组, 通过iNode索引值来访问一个iNode结构
- 磁盘块区, 用于存放磁盘块, 可以看做磁盘块的数组, 通过块索引值来访问一个磁盘块



文件系统需要和RAMDISK驱动程序交互来进行读写, 因此文件系统需要计算好每一个区域在RAMDISK中的偏移量, 以保证正确读写RAMDISK的内容.

2.1.3. 目录

分类是管理的一种有效手段, 随着文件数量的增加, 我们自然希望有一种能够对文件进行分类的方法, 目录的出现正是为了方便文件的管理.

实际上, 目录只是一种特殊的文件. 它是一张记录了当前目录文件信息的表, 每一个表项记录了该目录中一个文件的文件名或目录的名称和iNode索引:

```
typedef int inode_t;
struct DirEntry {
  char filename[32];
  inode_t inode;
};
```

引入目录后, 我们就可以实现一个树型结构的文件系统了: 首先文件系统有一个根目录"/", 根目录下有一些文件和子目录, 子目录下又有它的文件和子目录... 需要注意的是, 由于目录在文件系统中的特殊作用, 用户进程不应该拥有对目录进行任意修改的权限. 例如在Linux下, 若用户执行如下命令:

```
mkdir temp
echo "Hello world" > temp
```

`echo`命令的执行将会失败, 因为`temp`是一个目录, 不是普通文件. `echo`命令被拒绝的原因是很容易理解的, 因为随意对目录进行写操作将会破坏目录中记录, 导致文件表和目录下的文件产生不一致, 严重的话还可能会损坏整个文件系统. 因此, 我们有必要在iNode结构中添加一个反映文件类型的属性:

```
struct Inode {
  char filename[32];
  size_t size;
  int type;
  block_t index[15];
};
```

当type为PLAIN时, 代表该文件是普通文件; 当type为DIR时, 代表该文件是目录. type还可以是其它类型, 例如PIPE等.

此外, 目录应该由文件系统本身进行管理, 并且向外提供如下操作:

- 创建一个空文件, 只要申请一个空闲的iNode结构, 填写文件名和文件大小, 类型设为普通文件, 并把index设为无效, 最后需要在当前目录中增加记录新文件的表项, 以确保可以在当前目录中找到新文件
- 创建一个目录, 与创建普通文件类似, 但需要把类型设为目录. 需要注意的是, 新建的目录并不是空的, 它含有两个表项`.`和`..`, 分别代表目录本身和它的父目录. 因此新建目录的大小并不是0, 而是`2*sizeof(struct dir_entry)`, 并且需要申请相应的数据块, 往数据块中写入`.`和`..`两个表项, 并填写相应的index
- 删除文件或目录, 只要回收相应的iNode结构和数据块, 然后从父目录的表项中删除相应表项即可. 如果删除的是一个目录, 还需要递归地对目录中的文件进行删除, 否则文件系统将无法回收那些已经分配的iNode结构和数据块, 造成资源的泄漏
- 读出目录的内容, 只需要遍历目录中的文件表即可

2.1.4. 硬链接与删除文件

创建一个硬链接的系统调用原型为:

```
int link(char *oldfile, char *newfile);
```

它建立一个文件名为newfile, 目标为oldfile的硬链接.

要实现硬链接是十分简单的, 只需要在newfile所在的目录的文件表中新建一项文件名为newfile, iNode与oldfile的iNode相同的表项即可. 这样, 一个文件就可能有多个不同的文件名. 为了记录一个文件被引用的状态, 我们需要为iNode结构添加一个引用计数器的属性link, 每当创建一个硬链接的时候, 就要对oldfile的iNode结构的引用计数器增加1.

```
struct Inode {
  char filename[32];
  size_t size;
  int type;
  int link;
  block_t index[15];
};
```

与`link`作用相反的系统调用是`unlink`, 它用于删除文件:

```
int unlink(char *filename);
```

它将文件名为filename的iNode结构的引用计数器减小1, 当引用计数器为0时, 回收该文件的iNode结构和所占用的磁盘块.

> **对目录的硬链接**
>
> Linux不允许对目录创建硬链接, 你知道为什么吗? 如果允许对目录创建硬链接, 你能否举出一个影响系统工作的例子?

引入了目录之后, 文件的文件名都已经记录在文件的父目录中了, 因此iNode结构中的文件名就不必使用了. 而对于一个支持硬链接的文件系统, iNode结构中的文件名将会变得毫无意义, 因为此时iNode结构与文件名不再是一一对应的关系, 一个文件允许存在多个别名.

```
struct Inode {
  size_t size;
  int type;
  int link;
  block_t index[15];
};
```

2.1.5. 转移到磁盘

在RAMDISK中实现文件系统, 并确认文件系统正确工作之后, 现在可以考虑将整个文件系统转移到磁盘上, 实现文件的永久存储.

一个需要注意的地方是, 磁盘上已经存放了MBR和内核的可执行文件, 文件系统应该避开它们. 我们将整个文件系统存放在内核可执行文件之后的某个位置, 例如

```
size: sectors
0         1         201       203       205       207      463
-------------------------------------------------------------
          |         |         |         |         |        | user
   MBR    | kernel  | super   | iNode   | block   | iNode  | disk
          |         | block   | bitmap  | bitmap  |        | space
-------------------------------------------------------------
```

在真实的操作系统中, 内核是文件系统中的一个文件, 它的存放需要遵循文件系统的格式, 而不是像我们实验中超越了文件系统的存在, 在磁盘中连续存放. 系统引导的时候会先将一个小型文件系统加载到内存, 然后通过小型文件系统找到磁盘上的内核可执行文件, 再将内核载入内存. 我们的简化省去了这一过程, 从而不必关心其中涉及的细节问题.

另外, 磁盘的最小读写单位是一个扇区, 粒度比RAMDISK粗得多, 这意味着修改磁盘上的一个字节需要更大的代价: 首先需要从磁盘上读出该字节所在的整个扇区内容, 然后进行修改, 最后再将整个扇区写回到磁盘中. 读写磁盘的速度要远远低于读写内存的速度, 为了提高文件系统的性能, IDE中通常设有相应的缓存.

由于磁盘以扇区为单位进行读写, 我们最好让整数个iNode结构填满一个扇区, 而不是让某些iNode结构跨越了相邻的两个扇区, 以稍稍提升文件系统的性能. 例如

```
union Inode {
    uint8_t byte[128];
    struct {
        size_t size;
        int type;
        int link;
        block_t index[15];
    };
};
typedef union Inode Inode;
```

使得一个iNode结构占用128字节的空间. 事实上, 我们还可以在iNode结构中加入更多的信息, 例如拥有者, 读写权限, 修改日期等, 但这些功能不作为实验的要求, 有兴趣的同学可以探究如何实现它们.

超级块

文件系统的开头一般会有超级块, 超级块会记录文件系统的参数, 如iNode总数, 磁盘块总数, 每个区域的偏移量, 块大小等, 文件系统初始化的时候将会读取超级块中的内容.

2.2. 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]);
  }
}
```

2.3. 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`

2.4. 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. ~~初始化8253定时器（`initTimer`）~~
9. 配置好键盘映射表（`initKeyTable`）
10. 配置文件系统（`initFS`）
11. 从磁盘加载用户程序到内存相应地址（`loadUMain`）
12. 进入用户空间（`enterUserSpace`）
13. 调用库函数 `printf`

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

2.4.1. 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

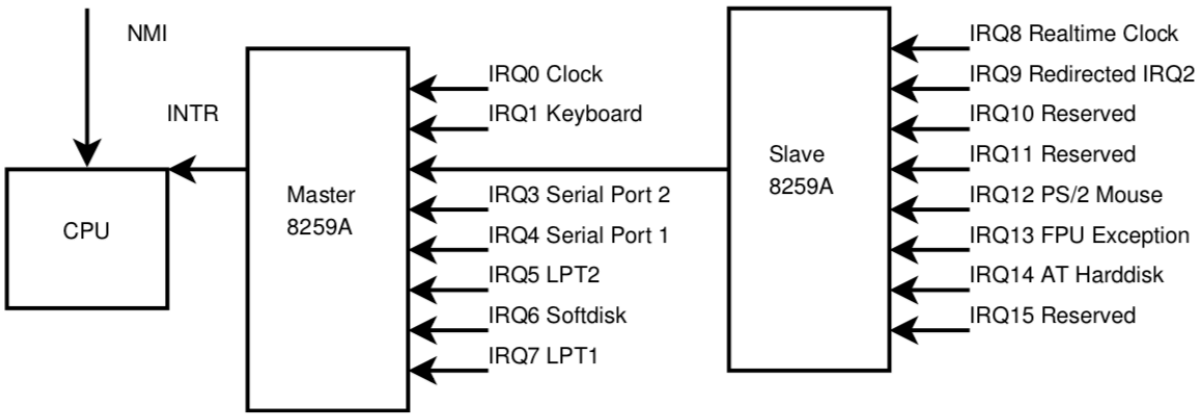
- **Fault**: 一种可被更正的异常, 一旦被更正, 程序可以不失连续性地继续执行, 中断程序返回地址为产生 Fault 的指令
- **Trap**: 发生 Trap 的指令执行之后立刻被报告的异常, 也允许程序不失连续性地继续执行, 但中断程序返回地址是产生 Trap 之后的那条指令
- **Abort**: Abort 异常不总是精确报告发生异常的位置, 它不允许程序继续执行, 而是用来报告严重错误.

2.4.2. 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。基于中断机制可以解决轮询处理硬件外设 I/O 时效率低下的问题。

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

2.5. 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 $0x80`

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：

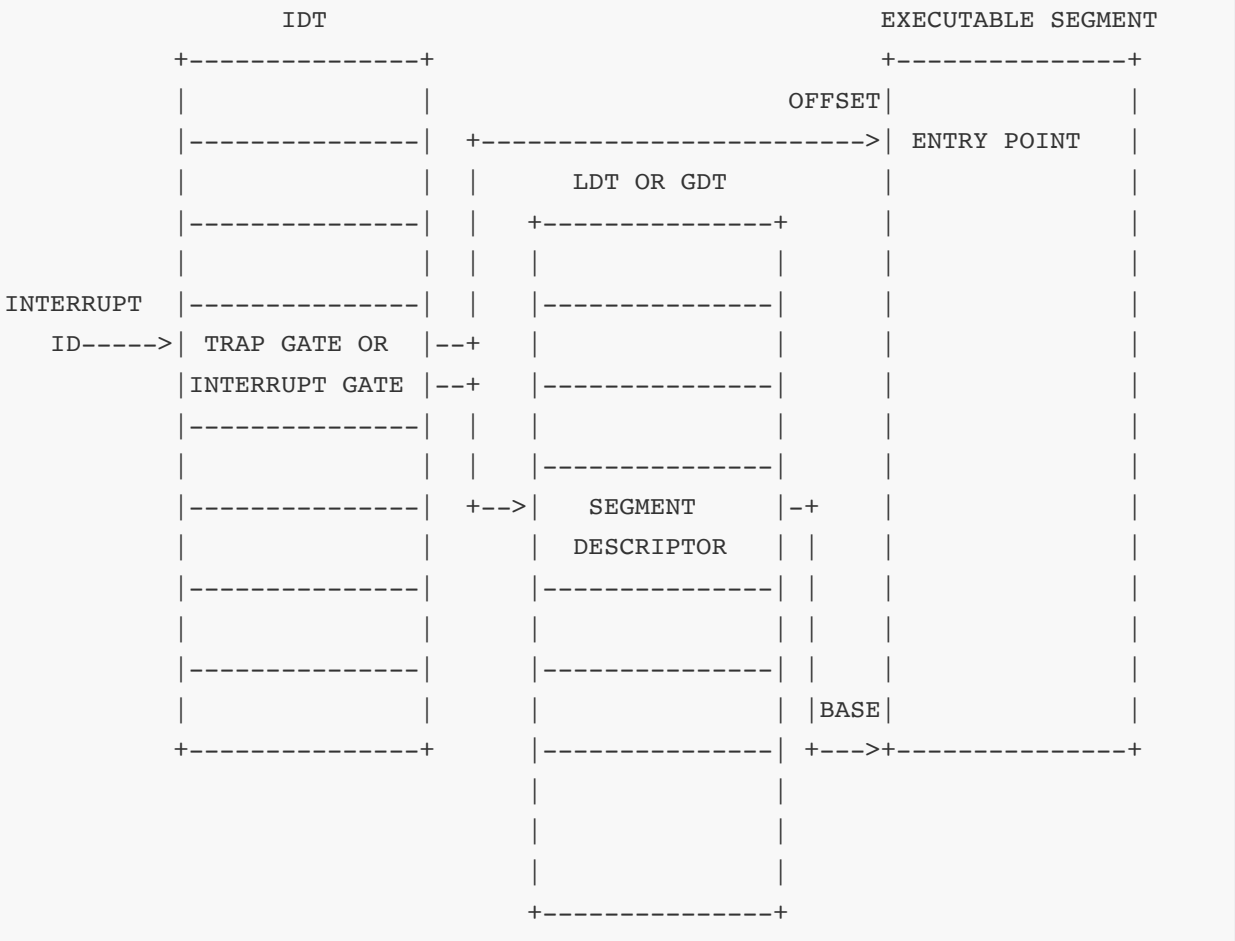
- 通过 `sti`，`cli` 指令设置 CPU 的 `EFLAGS` 寄存器中的 `IF` 位，可以控制对这些中断进行屏蔽与否
- 通过设置 8259A 芯片，可以对每个 IRQ 分别进行屏蔽

在我们的实验过程中，不涉及对IRQ分别进行屏蔽

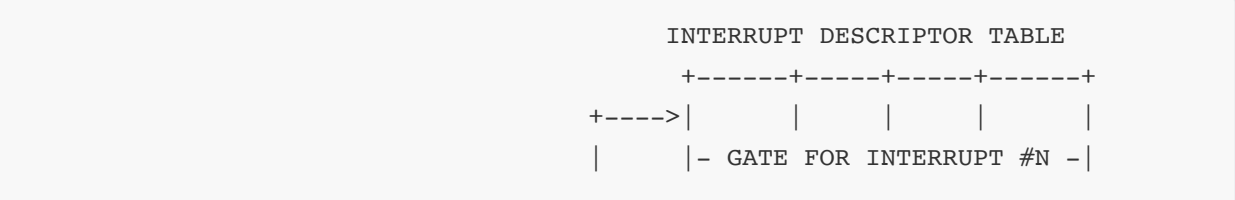
2.5.1. IDT

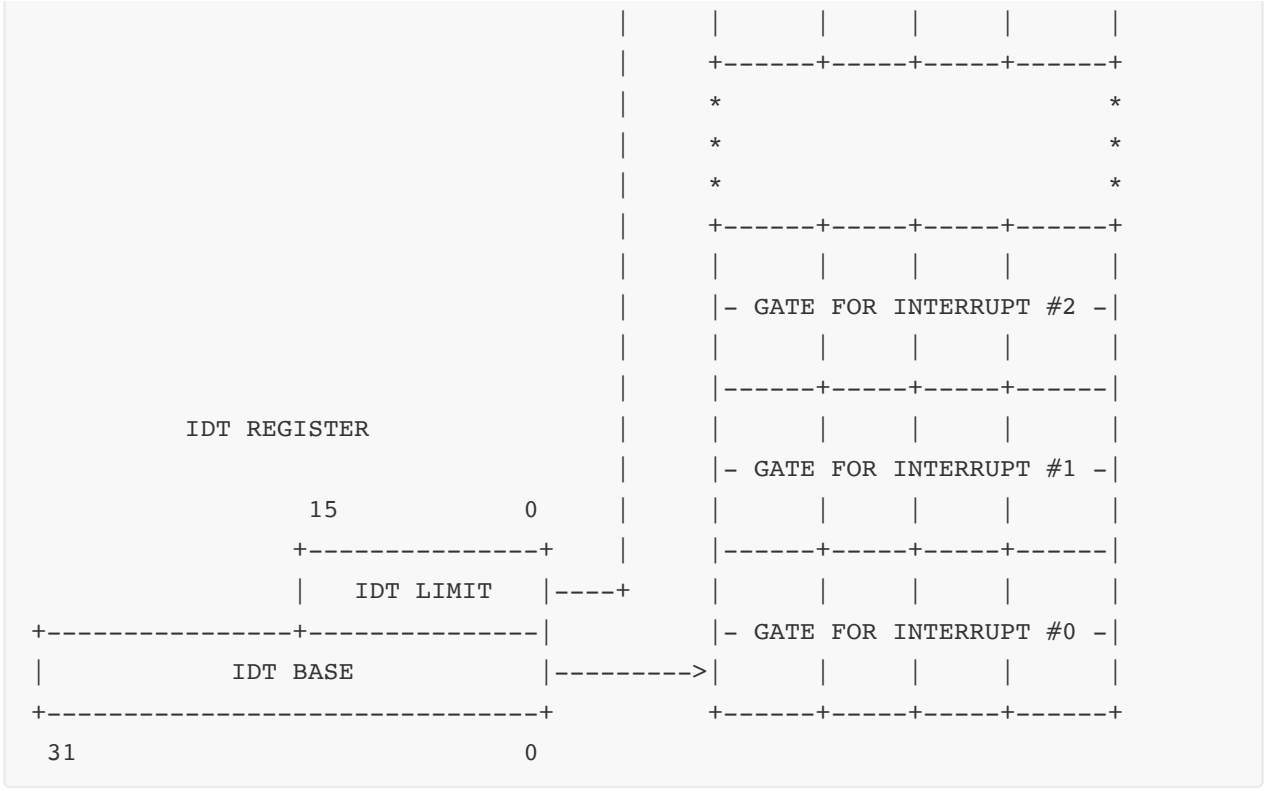
在保护模式下，每个中断(Exception，Interrupt，Software Interrupt)都由一个 8-Bits 的向量来标识，Intel 称其为中断向量，8-Bits表示一共有256个中断向量；与 256 个中断向量对应，IDT 中存有 256 个表项，表项称为门描述符(Gate Descriptor)，每个描述符占 8 个字节

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

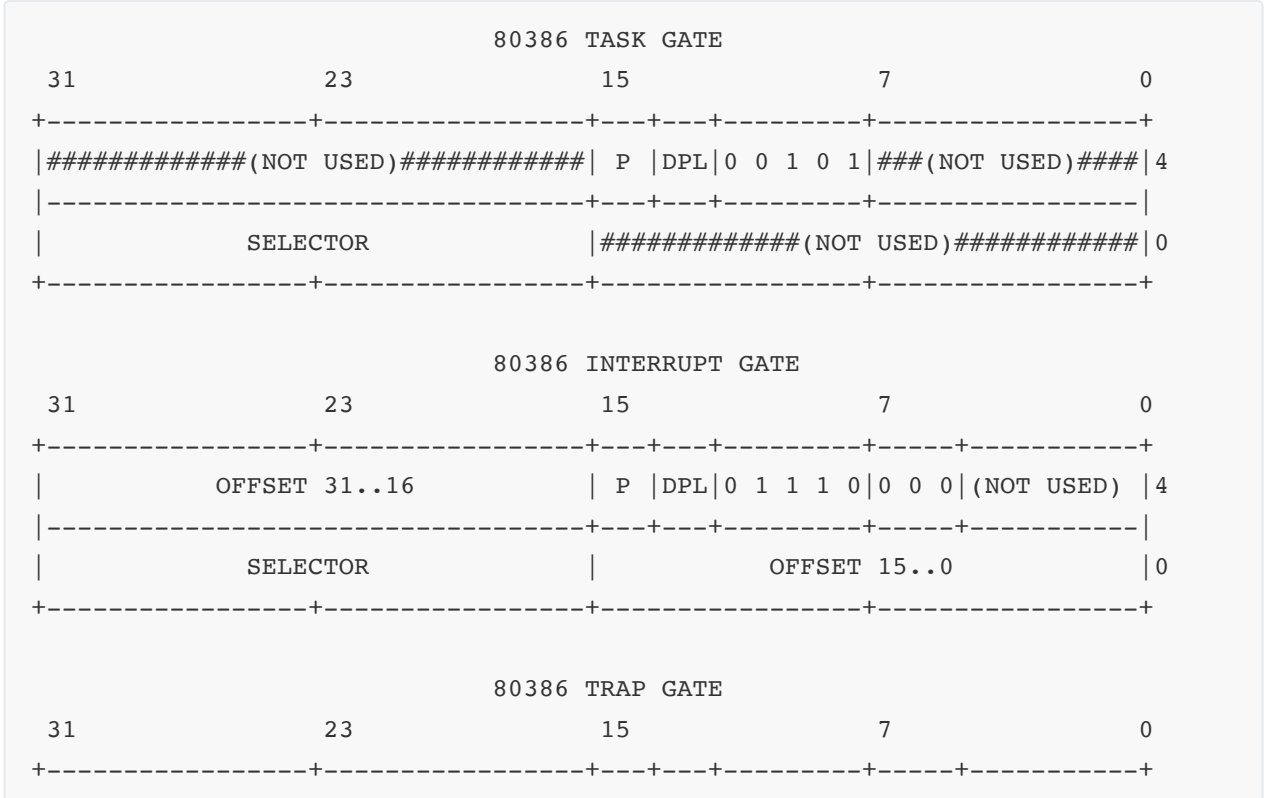




IDT中每个表项称为门描述符（Gate Descriptor），门描述符可以分为3种

- 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



```
|          OFFSET 31..16            | P |DPL|0 1 1 1 1|0 0 0|(NOT USED) |4
|-----------------------------------+---+---+---------+-----+-----------|
|             SELECTOR              |           OFFSET 15..0            |0
+-----------------+-----------------+-----------------+-----------------+
```

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

2.5.2. 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.

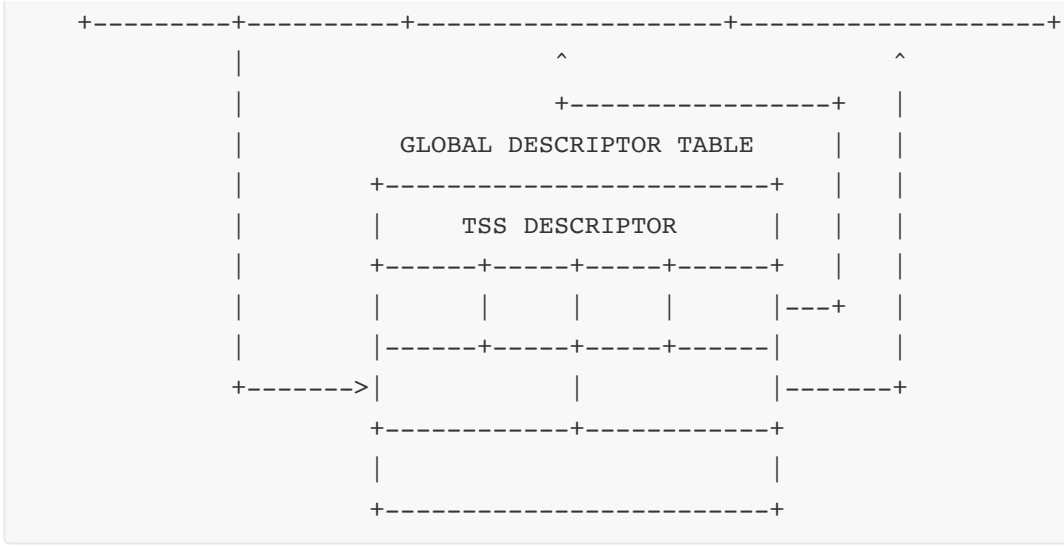
接下来的问题是, 哪些内容表征了A的状态? CPU又应该将它们保存到哪里去? 在IA-32中, 首先当是`EIP`(instruction pointer)了, 它指示了A在被打断的时候正在执行的指令; 然后就是`EFLAGS`(各种标志位)和`CS`(代码段, CPL). 由于一些特殊的原因, 这三个寄存器的内容必须由硬件来保存. 此外, 通用寄存器(GPR, general propose register)的值对A来说还是有意义的, 而进行中断处理的时候又难免会使用到寄存器. 但硬件并不负责保存它们, 因此我们还需要手动保存它们的值.

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

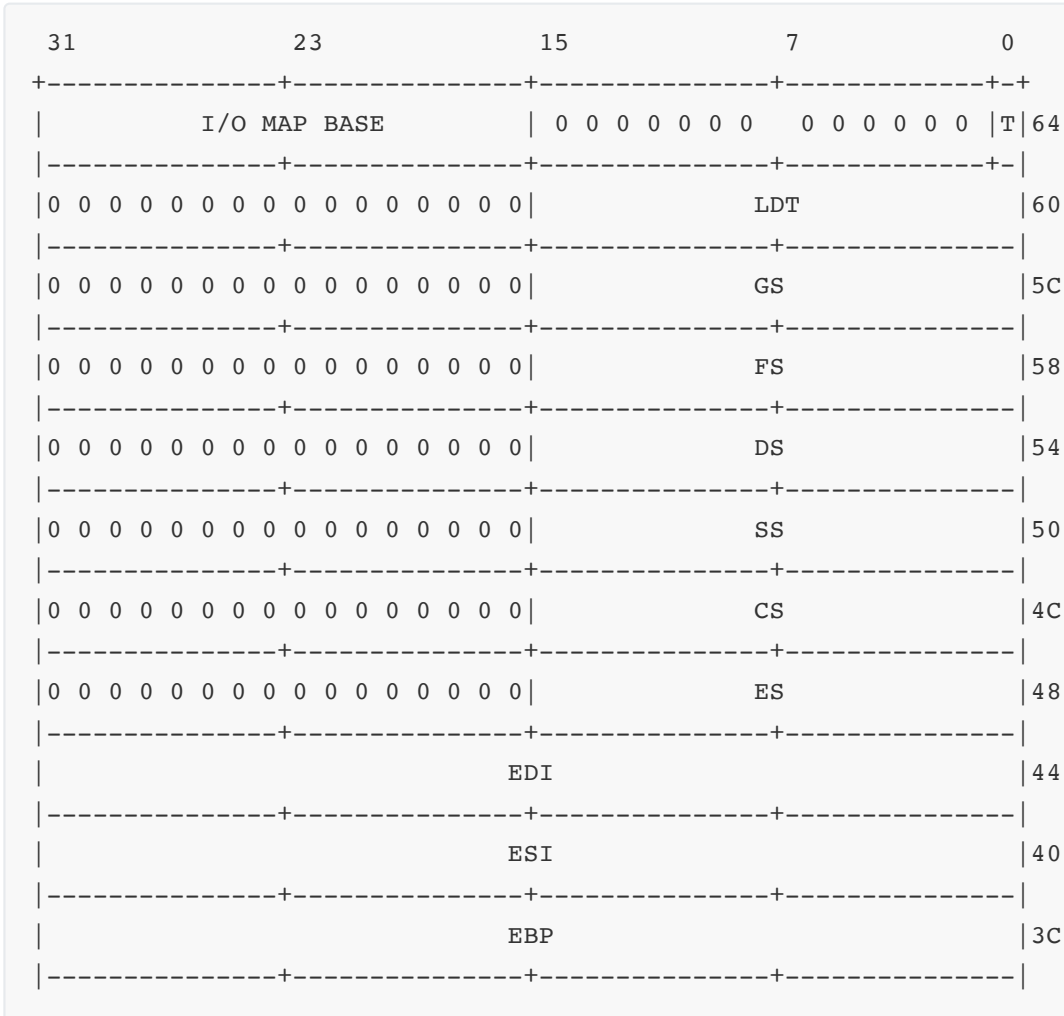
```
                      +-------------------------+
                      |                         |
                      |                         |
                      |                         |
                      |       TASK STATE        |
                      |        SEGMENT          |<---------+
                      |                         |          |
                      |                         |          |
                      |                         |          |
                      +-------------------------+          |
       16-BIT VISIBLE              ^                       |
          REGISTER                 |   HIDDEN REGISTER     |
   +--------------------+---------+----------+-------------+------+
TR |      SELECTOR      |      (BASE)        |       (LIMT)       |
```



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



```
|                               ESP                               |38
|---------------+---------------+---------------+---------------|
|                               EBX                               |34
|---------------+---------------+---------------+---------------|
|                               EDX                               |30
|---------------+---------------+---------------+---------------|
|                               ECX                               |2C
|---------------+---------------+---------------+---------------|
|                               EAX                               |28
|---------------+---------------+---------------+---------------|
|                             EFLAGS                              |24
|---------------+---------------+---------------+---------------|
|                     INSTRUCTION POINTER (EIP)                   |20
|---------------+---------------+---------------+---------------|
|                           CR3  (PDPR)                           |1C
|---------------+---------------+---------------+---------------|
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|              SS2              |18
|---------------+---------------+---------------+---------------|
|                               ESP2                              |14
|---------------+---------------+---------------+---------------|
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|              SS1              |10
|---------------+---------------+---------------+---------------|
|                               ESP1                              |0C
|---------------+---------------+---------------+---------------|
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|              SS0              |8
|---------------+---------------+---------------+---------------|
|                               ESP0                              |4
|---------------+---------------+---------------+---------------|
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|   BACK LINK TO PREVIOUS TSS   |0
+---------------+---------------+---------------+---------------+
```

> **ring3的堆栈在哪里?**
>
> 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?

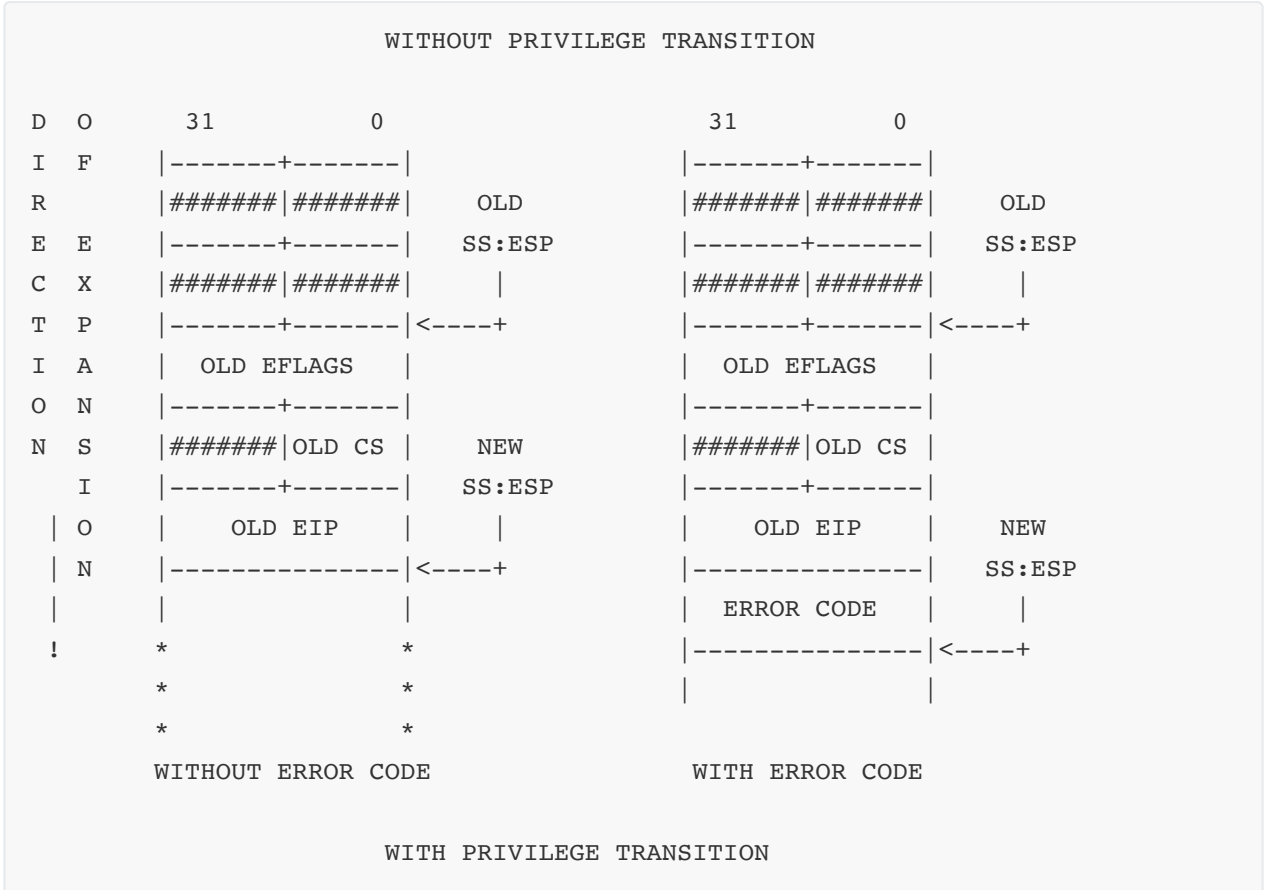
加入硬件堆栈切换之后, 中断到来/从中断返回的硬件行为如下

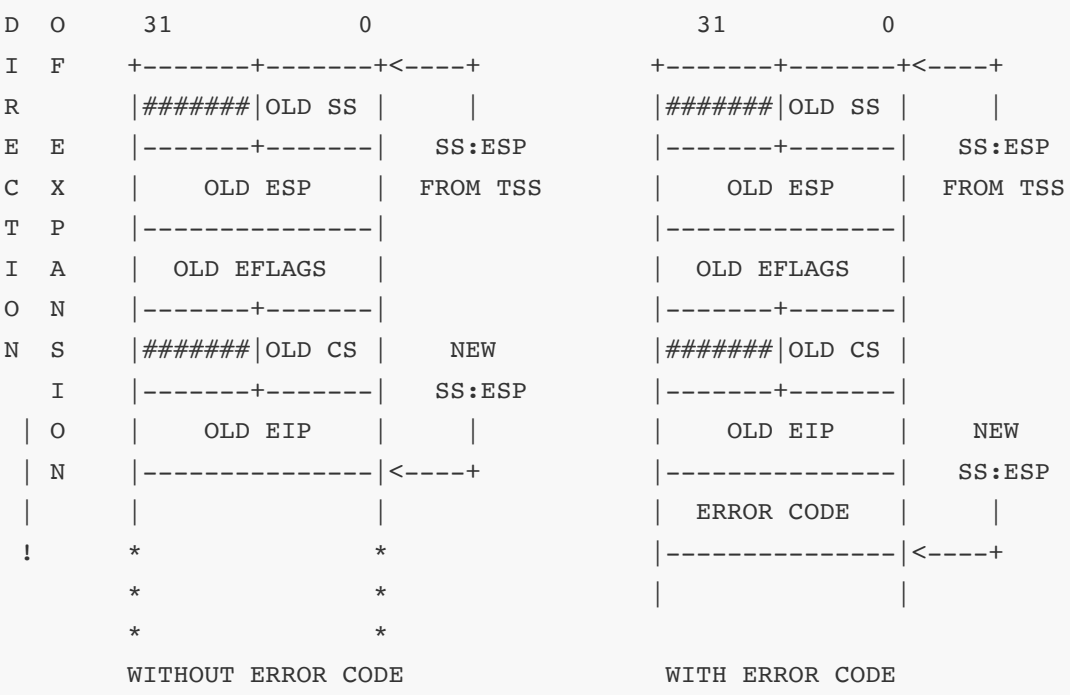
```
old_CS = CS
old_EIP = EIP
old_SS = SS
old_ESP = ESP
target_CS = IDT[vec].selector
target_CPL = GDT[target_CS].DPL
if(target_CPL < GDT[old_CS].DPL)
    TSS_base = GDT[TR].base
    switch(target_CPL)
        case 0:
            SS = TSS_base->SS0
            ESP = TSS_base->ESP0
```

```
        case 1:
            SS = TSS_base->SS1
            ESP = TSS_base->ESP1
        case 2:
            SS = TSS_base->SS2
            ESP = TSS_base->ESP2
    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





2.6. 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;
}
```

> **保存寄存器的旧值**
>
> 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` 指令压入内核堆栈的

2.7. 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

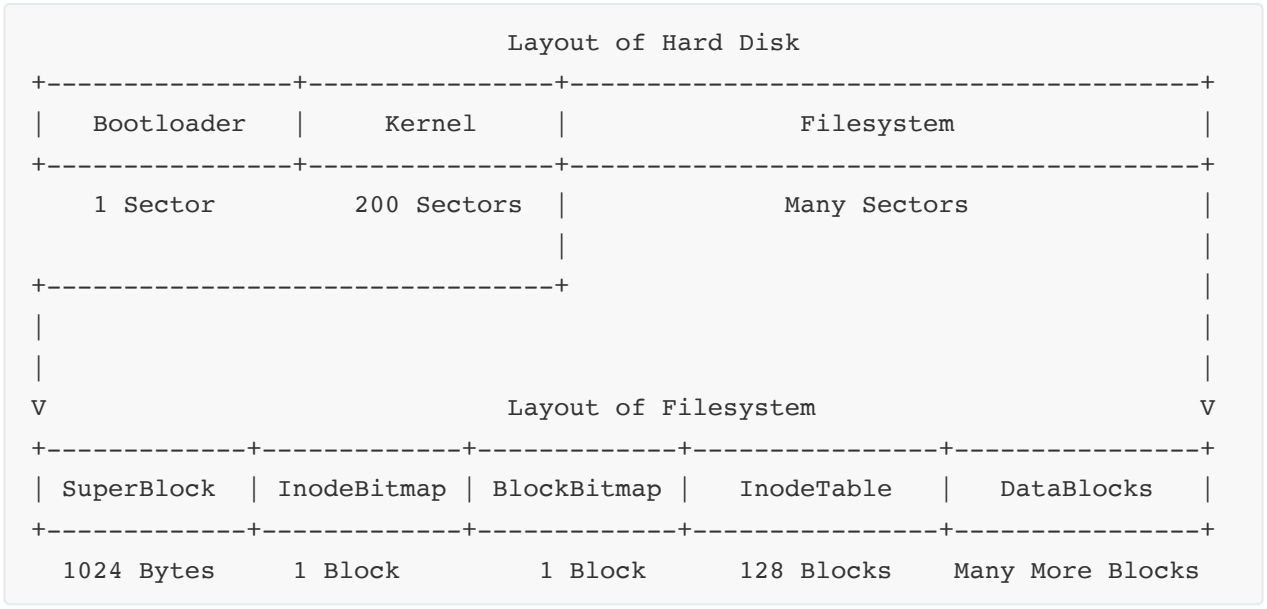
```
uint32_t getKeyCode() {
  uint32_t code = inByte(0x60);
  uint32_t val = inByte(0x61);
  outByte(0x61, val | 0x80);
  outByte(0x61, val);
  return code;
}
```

3. 解决思路

3.1. 格式化程序

格式化磁盘应该在我们运行自己的操作系统之前，也就是说格式化磁盘应该是在宿主机上进行的，我们将格式化磁盘的相关代码全部放在`lab2/utils/genFS`目录下。

往年的实验2用户程序放置在从201号扇区开始的磁盘文件中，内核初始化时需要从201号扇区顺序读取，实现对用户程序的加载；今年的实验我们要求将用户程序存储入文件系统中，可供参考的磁盘文件的布局如下所示



0号扇区为主引导扇区（MBR），1到200号扇区存储内核程序，201号开始的扇区按照文件系统的数据结构进行格式化，用户程序也依照文件系统的数据结构写入其中

文件系统的布局仿照MINIX文件系统，将扇区划分为Block进行管理（例如连续两个扇区作为一个Block）

文件系统头部为Meta Block，记录文件系统的元数据，依次包含Super Block、Inode Bitmap、Block Bitmap；Meta Block之后则为Inode Table以及真正存储通常文件以及目录文件的Data Block

Super Block中记录了整个文件系统的信息，例如总扇区数，Inode总数，Block总数，可用Inode数，可用Block数等

Inode Bitmap以及Block Bitmap则分别用于对Inode Table以及Data Block的使用情况进行记录

3.1.1. 文件系统数据结构

以下为本次实验使用的文件系统数据结构

```
union SuperBlock {
    uint8_t byte[SUPER_BLOCK_SIZE];
    struct {
        int32_t sectorNum;      // 文件系统中扇区总数
        int32_t inodeNum;       // 文件系统中inode总数
        int32_t blockNum;       // 文件系统中data block总数
        int32_t availInodeNum;  // 文件系统中可用inode总数
        int32_t availBlockNum;  // 文件系统中可用data block总数
        int32_t blockSize;      // 每个block所含字节数
        int32_t inodeBitmap;    // inode bitmap在文件系统的偏移，扇区为单位
        int32_t blockBitmap;    // block bitmap在文件系统的偏移，扇区为单位
        int32_t inodeTable;     // inode table在文件系统的偏移，扇区为单位
        int32_t blocks;         // data block在文件系统的偏移，扇区为单位
    };
};

union Inode {                           // Inode Table的表项
    uint8_t byte[INODE_SIZE];
    struct {
        int16_t type;                   // 该文件的类型、访存控制等
        int16_t linkCount;              // 该文件的链接数
        int32_t blockCount;             // 该文件的data block总数
        int32_t size;                   // 该文件所含字节数
        int32_t pointer[POINTER_NUM];   // data block偏移量，扇区为单位
        int32_t singlyPointer;          // 一级data block偏移量索引，扇区为单位
    };
};

union DirEntry {                    // 目录文件的表项
    uint8_t byte[DIRENTRY_SIZE];
    struct {
        int32_t inode;              // 该目录项对应的inode，编号从1开始，0表示空
        char name[NAME_LENGTH];     // 该目录项对应的文件名
    };
};
```

3.1.2. 文件系统目录结构

文件系统的目录结构如下所示，本次实验要求编写格式化程序对其进行构建

```
/
├── .             # 目录本身
├── ..            # 父目录（根目录的父目录为自身）
├── boot
│   ├── .         # 目录本身
│   ├── ..        # 父目录（指向根目录）
│   └── initrd    # 用户态初始化程序
└── usr
    ├── .
    └── ..
```

其中`/boot/initrd`为用户态初始化程序

看一下`genFS/main.c`如何构建文件系统

```
int main(int argc, char *argv[]) {
    char driver[NAME_LENGTH];
    char srcFilePath[NAME_LENGTH];
    char destFilePath[NAME_LENGTH];

    stringCpy("fs.bin", driver, NAME_LENGTH - 1);
    format(driver, SECTOR_NUM, SECTORS_PER_BLOCK);

    stringCpy("/boot", destFilePath, NAME_LENGTH - 1);
    mkdir(driver, destFilePath);

    stringCpy(argv[1], srcFilePath, NAME_LENGTH - 1);
    stringCpy("/boot/initrd", destFilePath, NAME_LENGTH - 1);
    cp(driver, srcFilePath, destFilePath);

    stringCpy("/usr", destFilePath, NAME_LENGTH - 1);
    mkdir(driver, destFilePath);

    stringCpy("/", destFilePath, NAME_LENGTH - 1);
    ls(driver, destFilePath);

    stringCpy("/boot", destFilePath, NAME_LENGTH - 1);
    ls(driver, destFilePath);

    stringCpy("/usr", destFilePath, NAME_LENGTH - 1);
    ls(driver, destFilePath);

    return 0;
}
```

可以看到是先使用`format`建立一个`fs.bin`文件做为文件系统载体，这一步还会建立一个根目录，完成后文件系统的目录结构如下：

```
/
├── .
└── ..
```

然后使用`mkdir`在根目录新建一个`boot`文件夹，完成后文件系统目录结构如下：

```
/
├── .
├── ..
└── boot
    ├── .
    └── ..
```

然后使用`cp`将通过`argv[1]`传递的宿主机上的文件拷贝到`boot`目录下，命名为`initrd`，完成后：

```
/
├── .
├── ..
└── boot
    ├── .
    ├── ..
    └── initrd
```

最后建立一个`usr`文件夹：

```
/
├── .
├── ..
├── boot
│   ├── .
│   ├── ..
│   └── initrd
└── usr
    ├── .
    └── ..
```

3.1.3. 从mkdir到cp

框架代码中已经实现了`format`、`ls`和`mkdir`，但是`cp`需要同学们实现。

仔细阅读`mkdir`的源码，想一想新建一个文件夹与新建一个文件并拷贝内容到文件的异同点，完成`cp`。

3.2. 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。

3.2.1. 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

3.2.2. 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

3.3. 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 = USEL(SEG_UDATA); //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 in lab2
  }
  updateCursor(displayRow, displayCol);
  //TODO take care of return value
}
```

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

3.4. 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);
}
```

为了方便使用, 你需要实现`%d`, `%x`, `%s`, `%c`四种格式转换说明符, 如果你不清楚它们的含义, 请查阅相关资料.

我们还为大家准备了`printf`的测试代码

```
  printf("printf test begin...\n");
  printf("the answer should be:\n");
  printf("###############################################################\n");
  printf("Hello, welcome to OSlab! I'm the body of the game. ");
  printf("Bootblock loads me to the memory position of 0x100000, and Makefile
also tells me that I'm at the location of 0x100000. ");
```

```
  printf("\\%%~!@#/(^&*()_+`1234567890-=..... ");
  printf("Now I will test your printf: ");
  printf("1 + 1 = 2, 123 * 456 = 56088\n0, -1, -2147483648, -1412567295,
-32768, 102030\n0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e\n");
  printf("##########################################################\n");
  printf("your answer:\n");
  printf("==========================================================\n");
  printf("%s %s%scome %co%s", "Hello,", "", "wel", 't', " ");
  printf("%c%c%c%c%c! ", 'O', 'S', 'l', 'a', 'b');
  printf("I'm the %s of %s. %s 0x%x, %s 0x%x. ", "body", "the game",
"Bootblock loads me to the memory position of", 0x100000, "and Makefile also
tells me that I'm at the location of", 0x100000);
  printf("\\%%~!@#/(^&*()_+`1234567890-=..... ");
  printf("Now I will test your printf: ");
  printf("%d + %d = %d, %d * %d = %d\n", 1, 1, 1 + 1, 123, 456, 123 * 456);
  printf("%d, %d, %d, %d, %d, %d\n", 0, 0xffffffff, 0x80000000, 0xabcdef01,
-32768, 102030);
  printf("%x, %x, %x, %x, %x, %x\n", 0, 0xffffffff, 0x80000000, 0xabcdef01,
-32768, 102030);
  printf("==========================================================\n");
  printf("Test end!!! Good luck!!!\n");
```

在你觉得你完成了`printf`的实现之后，将上述测试代码放入`lab2/app/main.c`中，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！

4. 代码框架

```
lab2-STUID                          #自行修改后打包(.zip)提交
├── lab2
│   ├── 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
│   │   │   │   └── utils.h
│   │   │   ├── common.h
│   │   │   ├── device
```

```
│   │   │   │   ├── disk.h
│   │   │   │   ├── keyboard.h
│   │   │   │   ├── serial.h
│   │   │   │   ├── timer.h
│   │   │   │   └── vga.h
│   │   │   ├── device.h
│   │   │   ├── fs
│   │   │   │   └── minix.h
│   │   │   ├── fs.h
│   │   │   ├── x86
│   │   │   │   ├── 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
│   │   │   ├── fs.c                #准备文件系统
│   │   │   ├── 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
│   │   │   ├── timer.c
│   │   │   └── vga.c
│   │   ├── lib
│   │   │   ├── abort.c
│   │   │   └── utils.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
│       ├── genFS                   #格式化程序
│       │   ├── Makefile
│       │   ├── data.h
│       │   ├── func.c
│       │   ├── func.h
│       │   ├── main.c
│       │   ├── types.h
│       │   ├── utils.c
│       │   └── utils.h
│       └── genKernel.pl            #生成内核程序
└── report
    └── 171220000.pdf
```

5. 相关资源

- US-QWERTY键盘扫描码

6. 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.
- **请大家在提交的实验报告中注明你的邮箱, 方便我们及时给你一些反馈信息.**
- **学术诚信**: 如果你确实无法完成实验, 你可以选择不提交, 作为学术诚信的奖励, 你将会获得10%的分数; 但若发现抄袭现象, 抄袭双方(或团体)在本次实验中得0分.
- 请你在实验截止前务必确认你提交的内容符合要求(格式, 相关内容等), 你可以下载你提交的内容进行确认. 如果由于你的原因给我们造成了不必要的麻烦, 视情况而定, 在本次实验中你将会被扣除一定的分数, 最高可达50%.
- 实验不接受迟交，一旦迟交按**学术诚信**给分.
- 其他问题参看`index.pdf`中的**作业规范与提交**一章
- **本实验最终解释权归助教所有**

截止时间：2020-3-30 23:59:55