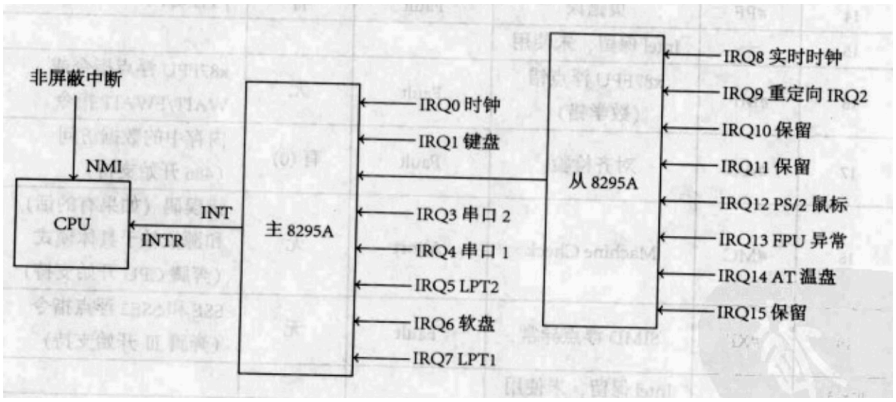
目 录

1 中断和异常的实现

1.1 设置 8259A

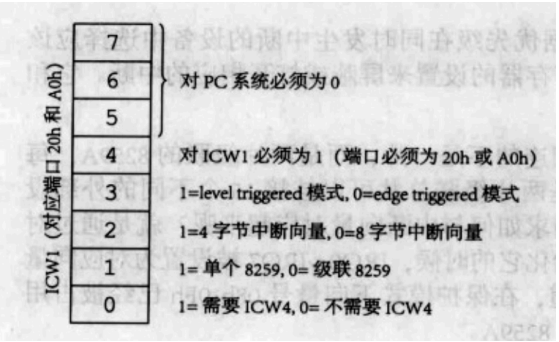
8259A 是中断机制中所有外围设备的一个代理，这个代理可以根据优先级在同时发生中断的设备中选择应该处理的请求。除此之外，还可以通过对 8259A 的寄存器的设置来屏蔽或打开相应的中断。可屏蔽外部中断与 CPU 是通过 8259A 连接起来的。8259A 与 CPU 的连接如下图所示：



由图可知，每一片 8259A 有 8 根中断信号线，两片级联的 8259A 可以挂接 15 个不同的外部设备。这些外部设备发出中断请求时，8259A 将其与相应的中断向量号对应起来。所以我们需要设置 8259A。

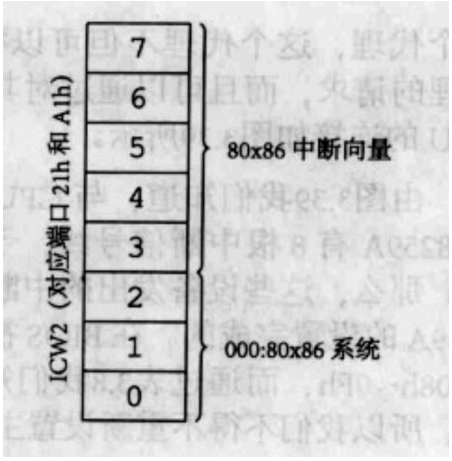
设置 8259A 的过程就是向其相应的端口写入特定的 ICW。主 8259A 的端口有 20h 和 21h，从 8259A 的端口有 A0h 和 A1h。ICW 全称是 Initialization Command Word，大小为一个字节。初始化 8259A 的过程如下：

- 首先往端口 20h 和 A0h 写入 ICW1。ICW1 的格式如图所示：

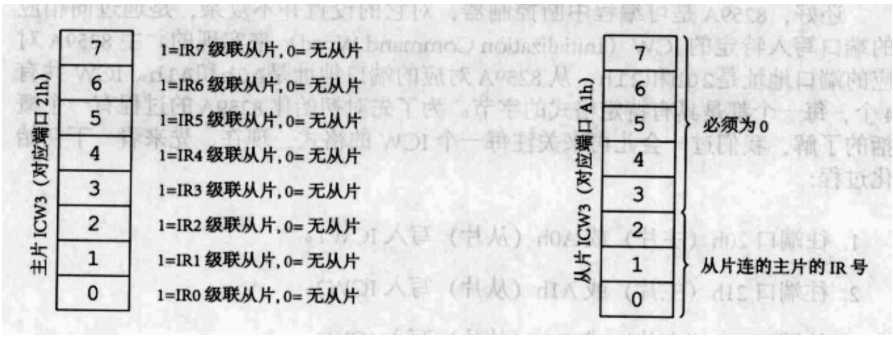


- 然后往端口 21h 和 A1h 写入 ICW2。主 8259A 和从 8259A 的 ICW2 内容可以不一样。写入 ICW2 时涉及与中断向量号的对应。比如，往主 8259A 写入 ICW2 时，如

果 ICW2 为 20h，那么 IRQ0 ～ IRQ7 就对应中断向量 20h ～ 27h。ICW2 的格式如图所示：



- 然后往端口 21h 和 A1h 写入 ICW3。主 8259A 的 ICW3 和从 8259A 的 ICW3 的格式不同。两个 ICW3 如图所示：



- 最后往端口 21h 和 A1h 写入 ICW4。ICW4 的格式如图所示：

实现代码如下所示：

```
; io_delay函数用于等待操作完成
io_delay:
    nop
    nop
    nop
    nop
    ret

Init8259A:
    ; 往端20h写入ICW1
    mov al, 011h
    out 020h, al
    call io_delay
    ; 往端A0h写入ICW1
    out 0A0h, al
    call io_delay
    ; 往端21h写入ICW2
    mov al, 020h
    out 021h, al
    call io_delay
    ; 往端A1h写入ICW2
    mov al ,028h
    out 0A1h, al
    call io_delay
    ; 往端21h写入ICW3
    mov al, 004h
    out 021h, al
    call io_delay
    ; 往端A1h写入ICW3
    mov al, 002h
    out 0A1h, al
    call io_delay
    ; 往端21h写入ICW4
    mov al, 001h
    out 021h, al
    call io_delay
    ; 往端A1h写入ICW4
    out 0A1h, al
    call io_delay
```

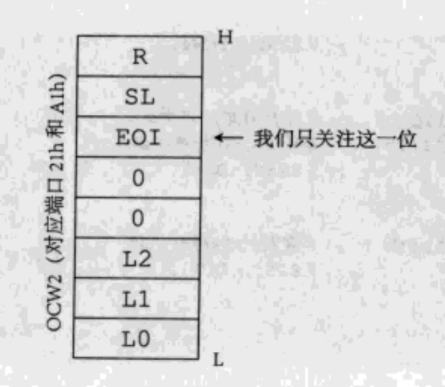
除了 ICW 字段，8259A 还接受 OCW 字段。OCW 全称为 operation control word。我们在两种情况下用到 OCW 字段。这两种情况是：

- 当屏蔽或打开外部中断时。此时端口 21h 或 A1h 写入 OCW1。OCW1 的格式如下图：

如果想屏蔽某一个外部中断，将对应的那一位设为 1 就可以了。实现代码如下：

```
        mov al, 11111111b
        out 0A1h, al
        call io_delay
```

- 发送 EOI 给 8259A 来通知它中断处理结束了，以便继续接收中断。此时往端口 20h 或 A0h 写 OCW2。OCW2 的格式如下图：



实现代码如下所示：

```
        mov al, 20h
        out 20h, al
```

1.2 建立 IDT

IDT 表和 GDT 表格式类似，它存放着中断门描述符、陷阱门描述符和任务门描述符。IDT 将每一个中断向量号和一个描述符对应起来。实现代码如下：

```
[SECTION .idt]
ALIGN 32
[BITS 32]
LABEL_IDT:
.01h: Gate SelectorCode32, SpuriousHandler, 0, DA_386IGate
; ...
IdtLen equ $ - LABEL_IDT
IdtPtr dw IdtLen - 1
       dd 0
```

加载 IDT 的代码如下：

```
xor eax, eax
mov ax, ds
shl eax, 4
add eax, LABEL_IDT
mov dword [IdtPtr + 2], eax

cli

lidt [IdtPtr]
```

1.3 实现一个中断

修改一下 IDT，将第 80h 号中断的中断处理函数改为 _UserIntHandler。实现代码如下所示：

```
[SECTION .idt]
ALIGN 32
[BITS 32]
LABEL_IDT:
; ...
.080h: Gate SelectorCode32, UserIntHandler, 0, DA_386IGate
; ...
IdtLen equ $ - LABEL_IDT
IdtPtr dw IdtLen - 1
       dd 0
; ...
[SECTION .s32]
[BITS 32]
_UserIntHandler:
UserIntHandler equ _UserIntHandler - $$
    mov ax, SelectorVideo
    mov gs, ax
    mov ah, 0Ch
    mov al, 'I'
    mov [gs:((80 * 0 + 70) * 2)], ax
    iretd
```

然后在 32 位代码段中添加如下代码，就可以实现一个中断。代码如下：

```
    call Init8259A
    int 080h
```

1.4 时钟中断试验

如果想打开时钟中断，一方面要打开外部中断，一方面要设计相应的中断处理程序。

首先打开外部中断，需要向 8259A 的 21h 或 A1h 写入相应的 OCW1，并且设置 IF 位为 1。时钟中断请求为 IRQ0。实现代码如下：

```
    ; 打开定时器中断
    mov al, 11111110b
    out 021h, al
    call io_delay
    ; 屏蔽从8159A的所有中断
    mov al, 11111111b
    out 0A1h, al
    call io_delay

    ret
```

在初始化 8259A 中，我们将 IRQ0 的中断向量号设置为 20h，所以需要在 IDT 的第 20h 项中写相应的时钟中断处理程序。

```
    [SECTION .idt]
    ALIGN 32
    [BITS 32]
    LABEL_IDT:
    ; ...
    .20h: Gate SelectorCode32, ClockHandler, 0, DA_386IGate
    ; ...

    [SECTION .s32]
    [BITS 32]
    ; 这个函数实现了将屏幕第0行、第70行的字符加一的功能
    _ClockHandler:
    ClockHandler equ _ClockHandler - $$
        inc byte [gs:((80 * 0 + 70) * 2)]
        ; 向端口20h写入OCW2，通知中断处理程序结束
        mov al, 20h
        out 20h, al
        iretd
```

下面是查看时钟中断效果的代码：

```
    ; 首先将屏幕第0行、第70行的字符设置为a
    mov ax, SelectorVideo
    mov gs, ax
    mov ah, 0Ch
    mov al, 'a'
    mov [gs:((80 * 0 + 70) * 2)], ax
    ; 初始化8259A
```

```
8     call Init8259A
9     ; 打开时钟中断
10    mov al, 11111110b
11    mov 021h, al
12    call io_delay
13    ; 设置IF位为1，打开外部中断
14    sti
15    ; 让程序陷入循环，可以查看每次时钟中断后屏幕上字符的变化
16    jmp $
```

1.5 几点需要注意的事

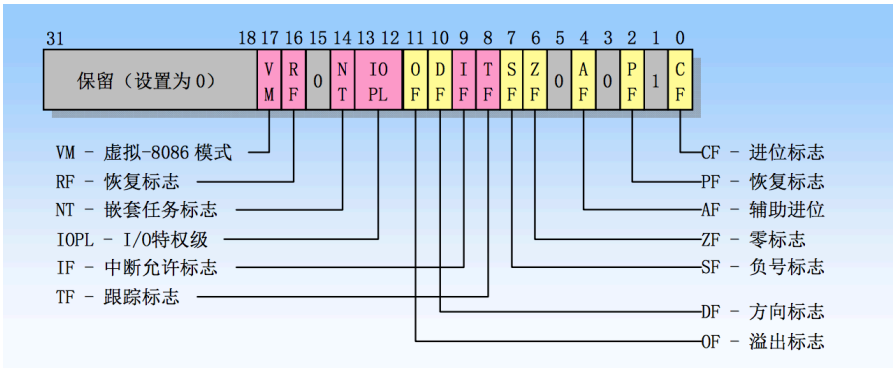
- 当中断产生时，大多会有特权级变换。通过中断门和陷阱门的中断相当于用 call 指令调用一个调用门，所以和第五次学习报告中特权级变换的内容一样。
- 中断或异常发生时，会和 call 指令一样进行压栈操作。需要注意的是，有些中断会产生出错码。如果有出错码，在 iretd 执行时，出错码是不会从堆栈中自动弹出的。所以在执行 iretd 之前，应该先将出错码从堆栈中清除掉。
- 中断门和陷阱门唯一的区别在于，中断门会对中断允许标志 IF 产生影响。由中断门向量引起的中断会复位 IF，在 iretd 指令执行后，会恢复 IF 位的原值。

2 保护模式下的 I/O

I/O 的控制权限是需要严格控制的，操作系统通过 IOPL 和 I/O 许可位图实现对 I/O 控制权限的限制。

2.1 IOPL

IOPL 字段位于 Eflags 寄存器的第 12、13 位。如下图所示：



操作系统将一些指令定义为 I/O 敏感指令，这些指令只有在 CPL≤IOPL 时才能执行，如果低特权级的任务试图执行这些指令将会引起一般性保护异常。I/O 敏感指令包括 in、ins、out、outs、cli 和 sti。

IOPL 字段是可以修改的，程序可以通过 popf 和 iretd 指令修改 IOPL 字段。只有当任务特权级为 0 时，popf 和 iretd 才可以成功修改 IOPL 的值。否则即使执行了指令，IOPL 也不会改变，不过也不会引起异常。

popf 指令还可以用来改变 IF 标志，只有当 CPL≤IOPL 时，才能成功修改 IF 标志，否则 IF 将维持原值，不会产生任何异常。

2.2 I/O 许可位图

在第五次学习报告中，我有实现过 TSS。其中代码有一处是“I/O 位图基址”。I/O 位图基址指向的就是 I/O 许可位图。I/O 许可位图的每一位用于表示一个字节的端口地址是否可用。如果该位为 0，表示此位对应的端口号可用，为 1 则代表不可用。I/O 许可位图的使用使得即便在同一特权级下不同任务也可以有不同的 I/O 访问权限。

I/O 许可位图就位于 TSS 段中，而 I/O 位图基址实际上是以 TSS 的地址为基址的偏移。如果 I/O 位图基址大等于 TSS 段界限，就表示 TSS 段中没有 I/O 许可位图。由于每个任务都有单独的 TSS，所以每个任务都有自己单独的 I/O 许可位图。

下面是一个任务中 I/O 许可位图的实现代码：

```
1    [SECTION .tss3]
2    LABEL_TSS3:
3        DW $ - LABEL_TSS3 + 2 ; 指向I/O许可位图
4    times 12 DB 0FFh ; 端口00h~5fh都不可用
5        DB 11111101b ; 端口60h~67h，只有端口61h可以用
6        DB 0FFh ; I/O许可位图结束标志,I/O许可位图必须以0FFh结尾
7    TSS3Len equ $ - LABEL_TSS3
```

3 linux 下的内存管理

3.1 页

内核用 struct page 结构表示系统中的每个物理页，这个结构在 mm_types.h 中定义。定义代码如下：

```
// 这个代码省去了联合结构体
struct page
{
    unsigned long flags;
    atomic_t _count;
    atomic_t _mapcount;
    unsigned long private;
    struct address_space* mapping;
    pgoff_t index;
    struct list_head lru;
    void* virtual;
}
```

下面是对 page 结构体中各个域的介绍：

- flag 域用来存放页的状态，flag 的每一位单独表示一种状态，所以它至少可以同时表示出 32 中不同的状态。这些标志在 page-flags.h 文件中定义，代码如下：

```
enum pageflags
{
    PG_locked, // 该页被锁住
    PG_error, // 此页发生了一个I/O错误
    PG_referenced, // used for page reclaim for anonymous pagecache
    PG_uptodate, // 表示该页的内容是否有效
    PG_dirty, // 该页为脏页
    PG_lru,
    PG_active,
    PG_slab,
    PG_owner_priv_1,
    PG_arch_1,
    PG_reserved, // 表示页永远不会被换出，也有可能不存在
    PG_private, // 表示该页含有文件系统中特定的数据
    PG_private_2,
    PG_writeback,
    PG_head,
    PG_swapcache,
    PG_mappedtodisk,
    PG_reclaim,
    PG_swapbacked,
    PG_unevictable,
    PG_mlocked,
    PG_uncached,
    PG_hwpoison,
    PG_young,
    PG_idle,
```

```
28          __NR_PAGEFLAGS,
29          PG_checked = PG_owner_priv_1,
30          PG_fscache = PG_private_2,
31          PG_pinned = PG_owner_priv_1,
32          PG_savepinned = PG_dirty,
33          PG_foreign = PG_owner_priv_1,
34          PG_slob_free = PG_private,
35          PG_double_map = PG_private_2,
36          PG_isolated = PG_reclaim
37      };
```

- _count 域存放在页的引用计数。
- virtual 域是页的虚拟地址，它就是页在虚拟内存中的地址。

需要知道的是，page 结构只是用于描述当前时刻在相关物理页中存放的东西。这个数据结构的目的在于描述物理内存本身，而并没有描述包含在其中的数据。内核使用这个数据结构来管理系统中所有的页。

3.2 区

系统中存在两种因为硬件缺陷而引起的内存寻址问题：

- 一些硬件只能用特定的内存地址来执行 DMA。
- 一些体系结构的内存的物理寻址范围比虚拟寻址范围大得多，导致有一些内存不能永久地映射到内核空间中。

为了解决这两个问题，内核把页分为了六个区：

- ZONE_DMA，这个区用于执行 DMA 操作。
- ZONE_DMA32，这个区同样用于执行 DMA 操作，只是这些页面只能被 32 位设备访问。
- ZONE_NORMAL，这个区包含的是能正常映射的页。
- ZONE_HIGHEM，这个区包含的页不能永久地映射到内核空间中。
- ZONE_MOVABLE
- ZONE_DEVICE

需要注意的是，区的划分是没有任何物理意义的，这只是内核为了管理页而采取的一种逻辑上的分组。linux 把系统的页划分为区，形成不同的内存池，然后根据用途进行分配。例如，当需要内存用于执行 DMA 操作，就可以从 ZONE_DMA 中按照请求的数目取出页。

下面是区的数据结构的定义，内核用它来管理系统中所有的区：

```
struct zone
{
    unsigned long watermark[NR_WMARK];
    unsigned long lowmem_reserve[MAX_NR_ZONES];
    struct per_cpu_pageset pageset[NR_CPUS];
    spinlock_t lock;
    struct free_area free_area[MAX_ORDER];
    spinlock_t lru_lock;
    struct zone_lru
    {
        struct list_head list;
        unsigned long nr_saved_scan;
    }lru[NR_LRU_LISTS];
    struct zone_reclaim_stat reclaim_stat;
    unsigned long pages_scanned;
    unsigned long flags;
    atomic_long_t vm_stat[NR_VM_ZONE_STAT_ITEMS];
    int prev_priority;
    unsigned int inactive_ratio;
    wait_queue_head_t *wait_table;
    unsigned long wait_table_hash_nr_entries;
    unsigned long wait_table_bits;
    struct palist_data *zone_pgdat;
    unsigned long zone_start_pfn;
    unsigned long spanned_pages;
    unsigned long present_pages;
    const char* name;
};
```

区的数据结构有 3 个重要的域，如下所示：

- watermark 数组。内核使用水位为每个内存区设置合适的内存消耗基准，watermark 数组持有该区水位所能达到的最小值、最低和最高水位值。
- lock 域。lock 是一个自旋锁，用于防止这个结构被并发访问。
- name 域。name 用于表示这个区的名字，内核启动期间将初始化这个值，三个区的名字分别为”DMA”、”Normal” 和”HighMem”。