Reference

一个操作系统的实现

x86汇编: 从实模式到保护模式

Intro

由于实验三完成内容过多, 实际上实验四需要完成的部分基本已经完成了, 所以我在这个使用报告详细说明中断处理机制及其实现, 以及键盘驱动的实现.

需要攻克的问题如下

- 键盘驱动及getchar实现
- printf实现

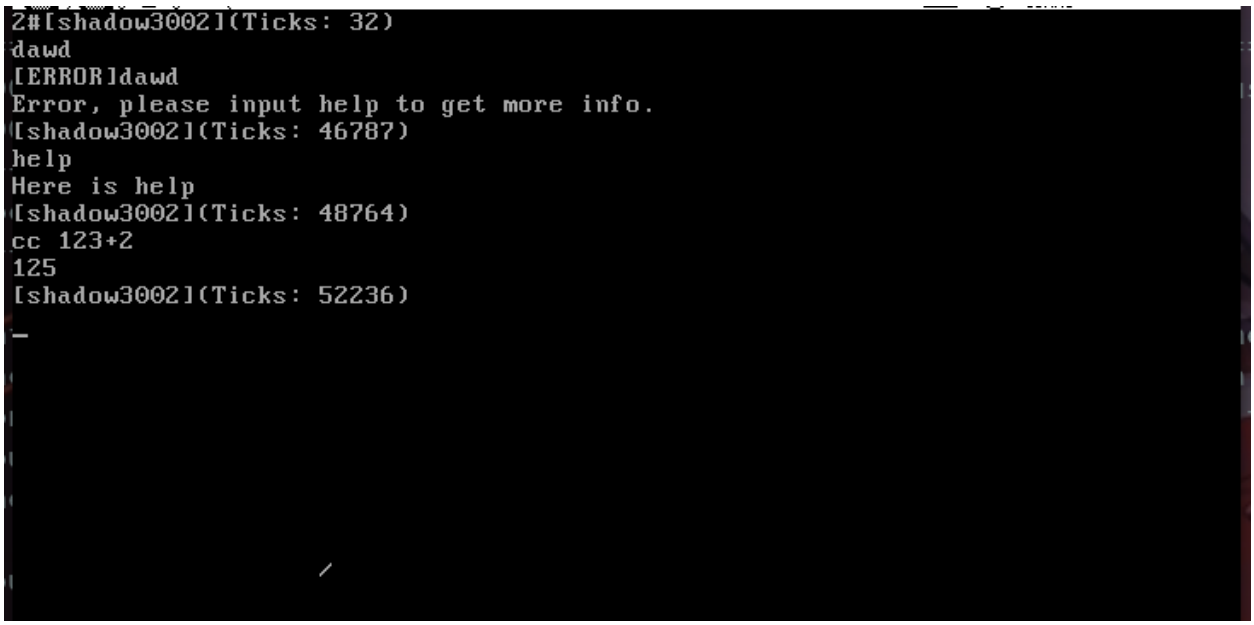
实验需求完成情况

老师需求

先按(Alt+F3)进入终端

- 风火轮

- 键盘handle



- 自定义中断

额外完成

- 在自定义中断的基础上实现了getchar

```
u8 sys_getchar(u32 u1, u32 u2, u32 u3, Process* p_proc)
{
    if (ring_length(&tty_table[p_proc->nr_tty].gb)) {return
pop_key(&tty_table[p_proc->nr_tty].gb);}
    return '\0';
}
```

- 以自定义中断为基础实现分时多进程

```
void schedule()
{
    Process*    p;
    int     greatest_ticks = 0;

    while (!greatest_ticks) {
        for (p = proc_table; p < proc_table + proc_num; p++) {
            if (p->state == READY) {
                if (p->ticks > greatest_ticks) {
                    greatest_ticks = p->ticks;
                    p_proc_ready = p;
                }
            }
        }

        if (!greatest_ticks)
            for (p = proc_table; p < proc_table + proc_num; p++)
                if (p->state == READY)
                    p->ticks = p->priority;
```

```
    }
}
```

- 以自定义中断为基础实现系统调用

```
;
========================================================================================
=
;                           void write(char* buf);
;
========================================================================================
=
write:
        mov     eax, _NR_write
        mov     edx, [esp + 4]
        int     INT_VECTOR_SYS_CALL
        ret

;
========================================================================================
=
;                           u8 getch();
;
========================================================================================
=
getch:
        mov     eax, _NR_getch

        int     INT_VECTOR_SYS_CALL
        ret

;
========================================================================================
=
;                   sendrec(int function, int src_dest, MESSAGE* msg);
;
========================================================================================
=
; Never call sendrec() directly, call send_recv() instead.
sendrec:
    mov eax, _NR_sendrec
    mov ebx, [esp + 4]  ; function
    mov ecx, [esp + 8]  ; src_dest
    mov edx, [esp + 12] ; p_msg
    int INT_VECTOR_SYS_CALL
    ret
```

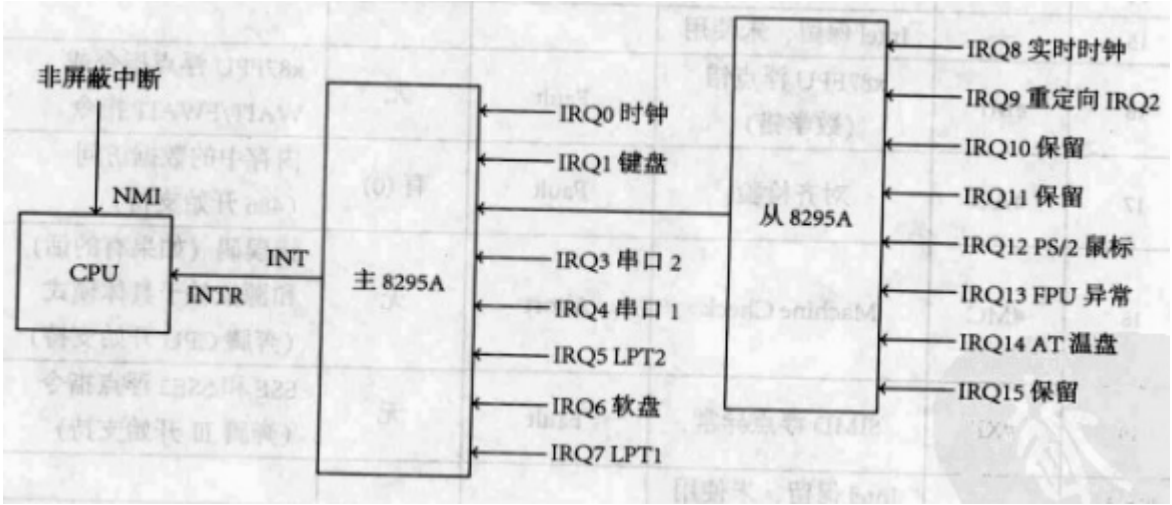
- 以自定义中断为基础实现进程通信

部分笔记及代码

8259a

8259a

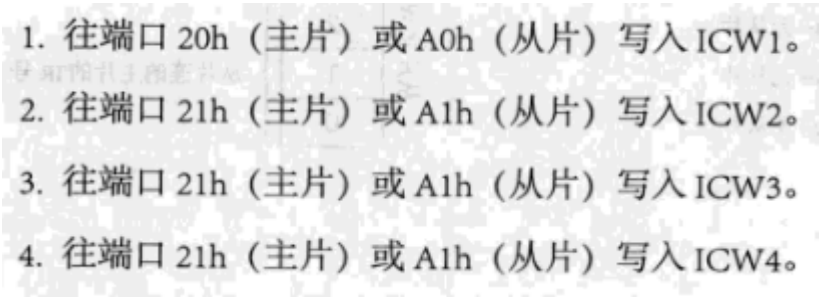
该芯片处理外部中断



BIOS初始化时, IRQ0-7被设置为08h到0Fh, 我们需要重新设置.

我们通过向端口写入特点ICW来设置.

初始化8259a




```
void init_8259A()
{
    out_byte(INT_M_CTL, 0x11);          // Master 8259, ICW1.
    out_byte(INT_S_CTL, 0x11);          // Slave  8259, ICW1.
    out_byte(INT_M_CTLMASK, INT_VECTOR_IRQ0);   // Master 8259, ICW2. 设置 '主8259' 的中
断入口地址为 0x20.
    out_byte(INT_S_CTLMASK, INT_VECTOR_IRQ8);   // Slave  8259, ICW2. 设置 '从8259' 的中
断入口地址为 0x28
    out_byte(INT_M_CTLMASK, 0x4);           // Master 8259, ICW3. IR2 对应 '从8259'.
    out_byte(INT_S_CTLMASK, 0x2);           // Slave  8259, ICW3. 对应 '主8259' 的 IR2.
    out_byte(INT_M_CTLMASK, 0x1);           // Master 8259, ICW4.
    out_byte(INT_S_CTLMASK, 0x1);           // Slave  8259, ICW4.

    out_byte(INT_M_CTLMASK, 0xFF);  // Master 8259, OCW1.
    out_byte(INT_S_CTLMASK, 0xFF);  // Slave  8259, OCW1.

    int i;
    for (i = 0; i < NR_IRQ; i++) {
```

```
            irq_table[i] = spurious_irq;
        }
    }
```

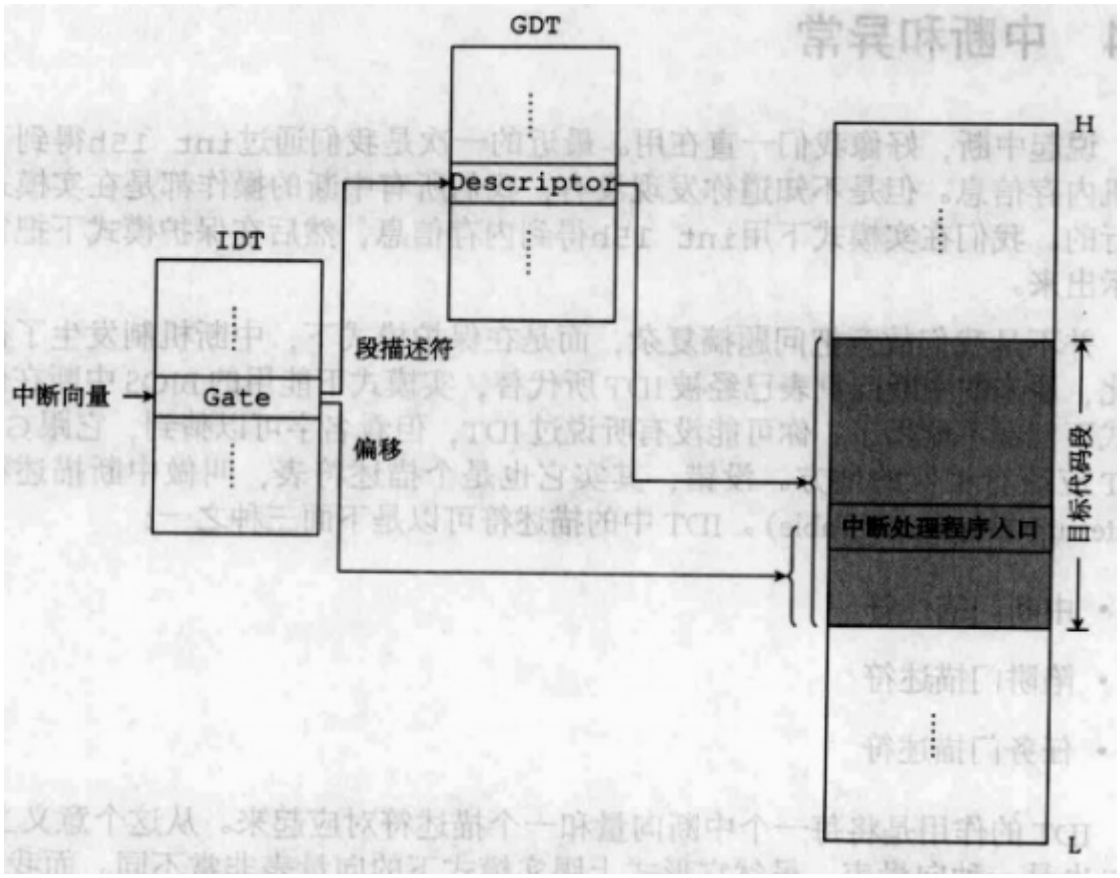
ICW格式

通过**OCW**屏蔽外部中断或发送EOI

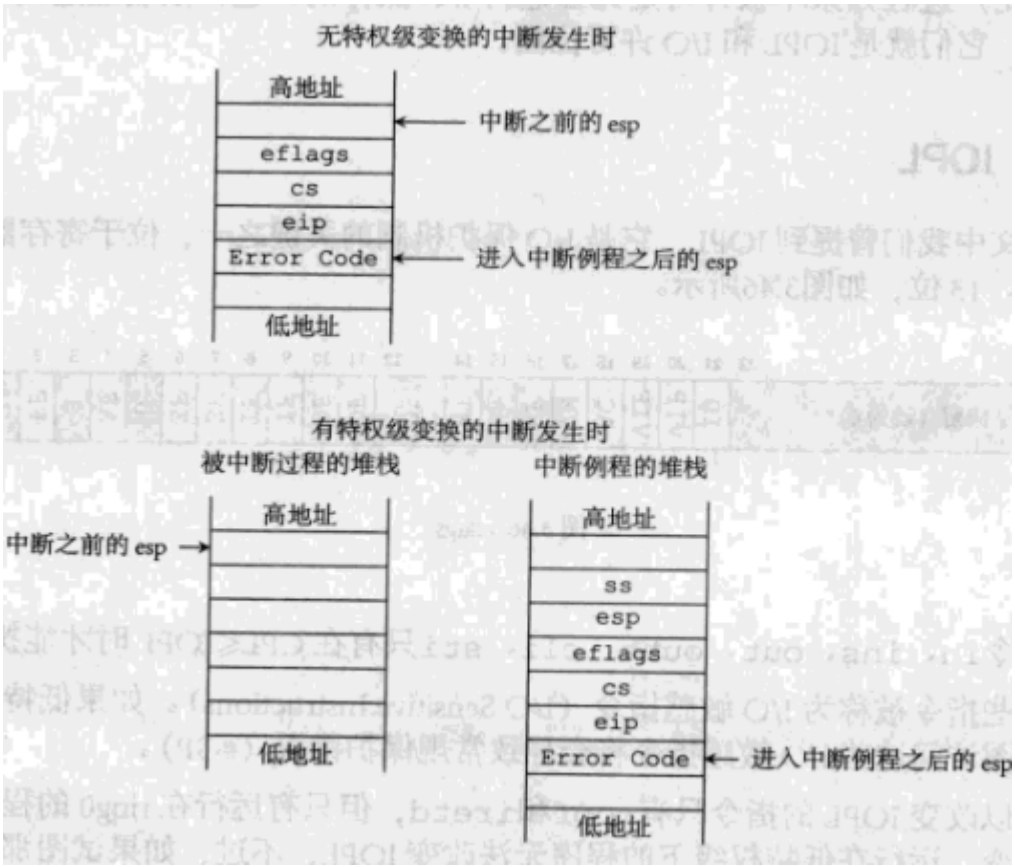
EOI: 结束中断处理, 实模式下每一次中断处理结束都需要发送一个EOI给8259a

中断处理过程

内部中断和外部中断使用统一中断号



中断时的堆栈变化



从中断返回时必须使用iretd, 与ret的区别是它同时会改变`eflags`

error code有时并不会有

所以需要按情况在iretd前处理sp

键盘驱动详解

键盘按下会中断产生`make`, 释放会产生`break`, 长按部分键比如空格会不断make和break.

所以我们需要

- 键盘驱动记录make, break
- 键盘驱动根据当前make, break, 发送字符给当前进程对应的tty
- tty处理, 存入二级缓冲ring buffer, 等待到`\n`, 就清空并发送到三级缓冲ring buffer
- 三级缓冲等待getchar

详细代码解释见实验三报告.

```
u8 sys_getchar(u32 u1, u32 u2, u32 u3, Process* p_proc)
{
    if (ring_length(&tty_table[p_proc->nr_tty].gb)) {return pop_key(&tty_table[p_proc-
>nr_tty].gb);}
    return '\0';
}
```

printf实现

- printf调用vsprintf处理出字符串

- vsprintf调用write系统调用
- write通过中断实现, 调用sys_write
- sys_write接受一个额外参数, 当前进程指针

可变参函数的实现原理是c/cpp调用约定: 调用者维护堆栈, 即调用者push参数, 被调用函数返回后, 调用函数维护`esp`

```
int printf(const char *fmt, ...)
{
    int i;
    char buf[256];

    va_list arg = (va_list)((char*)(&fmt) + 4); /*4是参数fmt所占堆栈中的大小*/
    i = vsprintf(buf, fmt, arg);
    buf[i] = '\0';
    write(buf);

    return i;
}
```

printf调用了vsprintf, 由vsprintf根据字符串判断多的参数的数目和类型

```
int vsprintf(char *buf, const char *fmt, va_list args)
{
    char*   p;

    va_list p_next_arg = args;
    int m;

    char    inner_buf[STR_DEFAULT_LEN];
    char    cs;
    int align_nr;

    for (p=buf;*fmt;fmt++) {
        if (*fmt != '%') {
            *p++ = *fmt;
            continue;
        }
        else {      /* a format string begins */
            align_nr = 0;
        }

        fmt++;

        if (*fmt == '%') {
            *p++ = *fmt;
            continue;
        }
        else if (*fmt == '0') {
            cs = '0';
            fmt++;
        }
```

```
        else {
            cs = ' ';
        }
        while (((unsigned char)(*fmt) >= '0') && ((unsigned char)(*fmt) <= '9')) {
            align_nr *= 10;
            align_nr += *fmt - '0';
            fmt++;
        }

        char * q = inner_buf;
        memset(q, 0, sizeof(inner_buf));

        switch (*fmt) {
        case 'c':
            *q++ = *((char*)p_next_arg);
            p_next_arg += 4;
            break;
        case 'x':
            m = *((int*)p_next_arg);
            i2a(m, 16, &q);
            p_next_arg += 4;
            break;
        case 'd':
            m = *((int*)p_next_arg);
            if (m < 0) {
                m = m * (-1);
                *q++ = '-';
            }
            i2a(m, 10, &q);
            p_next_arg += 4;
            break;
        case 's':
            strcpy(q, (*((char**)p_next_arg)));
            q += strlen(*((char**)p_next_arg));
            p_next_arg += 4;
            break;
        default:
            break;
        }

        int k;
        for (k = 0; k < ((align_nr > strlen(inner_buf)) ? (align_nr -
strlen(inner_buf)) : 0); k++) {
            *p++ = cs;
        }
        q = inner_buf;
        while (*q) {
            *p++ = *q++;
        }
    }

    *p = '\0';
```

```
    return (p - buf);
}
```

```
write:
        mov     eax, _NR_write
        mov     edx, [esp + 4]
        int     INT_VECTOR_SYS_CALL
        ret
```

sys_write通过进程指针写入正确console

```
int sys_write(u32 u1, u32 u2, char* s, Process* p_proc)
{
    char * p = s;
    char ch;

    while ((ch = *p++) != 0) {
        if (ch == MAG_CH_PANIC || ch == MAG_CH_ASSERT)
            continue; /* skip the magic char */

        out_char(tty_table[p_proc->nr_tty].p_console, ch);
    }

    return 0;
}
```

这里因为系统调用的参数固定为四个, 所以填充了无用参数

心得体会

在系统调用设计时为了方便重用代码只能固定参数大小和数量

处理中断嵌套很麻烦, 进入中断处理程序还需要额外开启, 关闭中断, 实现中断嵌套

特权级转移

- ring0 -> ring3, 通过调用门
- ring3 -> ring0, 通过TSS获取内核栈