操作系统实验七

前言

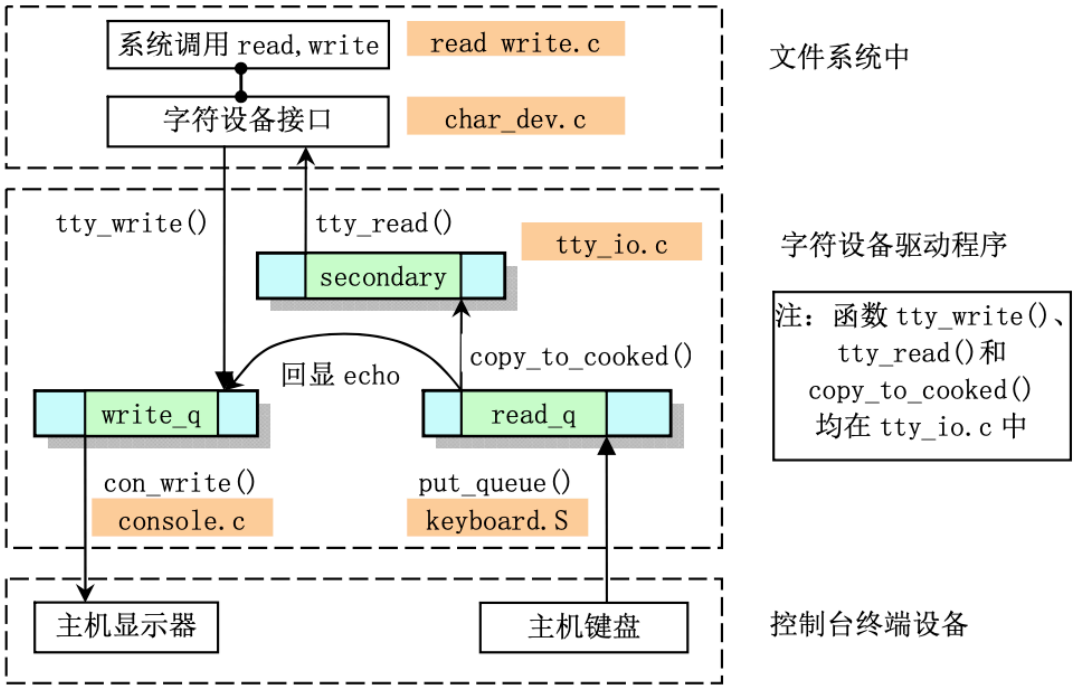
这个实验实际上是我在完成第三个实验之后完成的，因为第四个现在还不怎么会，等再研究研究，之后一看这个实验难度只是标了两颗星，想应该比较容易而且比较独立，实际上还是要分析这看一看，之后才实现的。

还是先来说一下环境的问题，这个看似很完美的环境（实验二的时候就不完美了），现在又出了一个小问题，在Mac OS 13的时候吧还是什么时候，苹果终于更新了其传统的HFS+文件系统，还成了Apple File System，也就是APFS，速度各方面都提升很多，但是居然有个问题，虽然Mac说我这是unix系统，但是APFS默认，拿到打开电脑，默认的APFS是不区分大小写的，如果要打开的话还得用些手段，而且很多应用还不支持改成大小写敏感的程序，这就尴尬了啊，居然不区分大小写，而这次实验中，非常重要的一个文件，`keyboard.S`，注意了，那个S是大写的，说是gcc的某种模式，可以在这个汇编程序中引用C宏还是什么的，但是有个神秘的地方，在编译之后，gcc会出现一个我看了就是没有基本没有注释和C语言宏的`keyboard.s`汇编文件，Linux等系统上这两个是不一样的文件，但是在默认APFS这里，boom，重复了，就会编译失败，这个问题实际上早就有了，只是一直忽略，因为这个实验文件实际上是我从docker里面的oslab直接拷贝出来的，拷贝过程中就报了文件已经存在警告，我没注意，现在才发现有这样的问题。

之后如何解决呢，其实可以在dockerLinux里面先编译一遍，之后在对应的文件夹里面就会有一个`keyboard.s`小写的文件，之后在Mac这里把现在文件夹下面的如果有的这个keyboard文件都删除掉，之后把docker Linux下面的这个小写的文件拷贝到Mac那里，编译，就成功了，我之后认真一看一眼有啥区别，发现就是把一大堆注释删除了，宏也没了，剩下的一模一样，也不影响实验，但是就是有这样的问题，如果你在Linux下面实验肯定不会有这样的问题，现在越来越觉得Linux才是写这次实验的最佳平台，只是我不想虚拟机或者物理机跑。

字符显示的控制

Linux这种系统，万物皆文件，实际上输出到终端，也算是输出到一种文件，先放解析书中的一幅图：



这就是如何将我主机键盘输入的内容最后通过处理在显示器上显示的整个过程，可以看到，最上面的一层是文件系统中，也就是在代码的fs文件夹下面，这也就是说其还是需要文件系统的支持。

流程上来看，和嵌入式学的按键中断是个差不多的东西，不过在单片机上面是不会走文件系统这一层的，会直接调用屏幕硬件，在Linux这里首先是主机键盘输入，键盘按键在这里按下一个键就是一个中断，之后调用按键中断程序，值得说的是，在Linus应该用的是AT机上，按下和弹起是两个状态，应该是分别用高4位和低4位表示，这里就提一下，这部分的主要工作也就是在keyboard.S中实现，这里先看一下这个文件的一部分，也就是这次实验需要改变功能的F12键是个什么东西：

```
# 这里是这次实验的重要地方，将功能键扫描码转换成转义字符并且存放到读队列中
func:
    cmpb $0x58,%al # 判断是不是F12
    jne other_func # 不是的话继续执行
    pushl %eax
    pushl %ecx
    pushl %edx
    call change_tty_hide
    popl %edx
    popl %ecx
    popl %eax
    jmp end_func /* 不再执行其他操作 */
other_func:
    pushl %eax
    pushl %ecx
    pushl %edx
    call show_stat # 调用显示各个任务状态函数，sched.c 第37行
    popl %edx
    popl %ecx
    popl %eax
```

```
    subb $0x3B,%al # 键F1的扫描码是0x3B，al寄存器就是功能键索引号
    jb end_func
    cmpb $9,%al # 功能键是F1-10吗
    jbe ok_func
    subb $18,%al # 是F11和F12吗，F11和F12的扫描码是0x57,0x58
    cmpb $10,%al
    jb end_func
    cmpb $11,%al
    ja end_func
ok_func:
    cmpl $4,%ecx # 每个功能键需要存放4个字符序列
    jl end_func
    movl func_table(,%eax,4),%eax # 取出功能键对应的字符序列
    xorl %ebx,%ebx
    jmp put_queue
end_func:
    ret
# 功能键发送的扫描码
func_table:
    .long 0x415b5b1b,0x425b5b1b,0x435b5b1b,0x445b5b1b
    .long 0x455b5b1b,0x465b5b1b,0x475b5b1b,0x485b5b1b
    .long 0x495b5b1b,0x4a5b5b1b,0x4b5b5b1b,0x4c5b5b1b
```

这里我已经修改过了，之后再说我修改了什么，keyboard.S的功能实际上是单纯的，它会忠实的把硬件的信号上传到上一层，至于想功能键这种东西，实际上是转化为转移字符，比如F1是`ESC [[ A`之后A，B，C这样类推，这里所以实际上是上报了4个字符，而且这里也给出了调用函数的方法，最初的文件，这里会先保存状态，之后调用执行显示当前所有任务函数这个函数，这里就给出一个思路就是也模仿这个地方，当我们判断到又0x58也就是F12的扫描码的时候，就调用一个函数，将当前的状态设置一下，之后让输入的文字都变成*。

之后现在还要说明的是，实际上系统会维护两个队列，一个是原始字符的，也就是说一堆码，一大串数字，这是不能直接就输出出去的，比如我输入的实际上是一个不可见字符，系统怎么办，输入的长度很长要换行怎么办，还有些神奇的玩意，比如\n换行，\v垂直方向换一行（我也不知道怎么描述一下），这就需要另一个队列，之后通过copy_to_cooked()这个函数来处理一下，cook烹饪也是很形象了，这个地方也可以入手修改，实现目标，之后看图，还有个是否回显，实际上我们也知道Linux系统在输入密码等等的时会不显示东西，就是靠这个信号，这个地方可以想办法修改成F12修改这个状态，之后设置不回显都显示成*。

在生成第二条字符串队列之后，就会传输到文件系统中，最后其实无论是写文件还是输出到终端，都是要调用系统中的read_write中的sys_write系统调用函数，而在这个函数中的文件写是file_write函数，之后这个函数在file_dev.c中是下面这个样子：

```
int file_write(struct m_inode *inode, struct file *filp, char *buf, int
count)
{
    off_t pos;
    int block, c;
    struct buffer_head *bh;
    char *p;
```

```
    int i = 0;

    /*
 * ok, append may not work when many processes are writing at the same
time
 * but so what. That way leads to madness anyway.
 */
    if (filp->f_flags & O_APPEND)
        pos = inode->i_size;
    else
        pos = filp->f_pos;
    while (i < count)
    {
        if (!(block = create_block(inode, pos / BLOCK_SIZE)))
            break;
        if (!(bh = bread(inode->i_dev, block)))
            break;
        c = pos % BLOCK_SIZE;
        p = c + bh->b_data;
        bh->b_dirt = 1;
        c = BLOCK_SIZE - c;
        if (c > count - i)
            c = count - i;
        pos += c;
        if (pos > inode->i_size)
        {
            inode->i_size = pos;
            inode->i_dirt = 1;
        }
        i += c;
        //* 其实本质上向什么地方输出都会调用这里，万物皆文件
        //* 向中断输出也是文件的一种，这里用三目表达式写简单一点
        while (c-- > 0)
            *(p++) = (tty_hide == 0) ? get_fs_byte(buf++) : '*', buf++;
        brelse(bh);
    }
    inode->i_mtime = CURRENT_TIME;
    if (!(filp->f_flags & O_APPEND))
    {
        filp->f_pos = pos;
        inode->i_ctime = CURRENT_TIME;
    }
    return (i ? i : -1);
}
```

所以实验中说的，要是修改成文件输出也变成*，就在这个地方修改，这里我修改了一下，这样文件输出我改了，去掉文件输入就不会是*了。

之后向后看，到终端控制中的函数，如果只修改这个地方的话就可以实现只修改终端的输出，下面是部分con_write函数：

```
void con_write(struct tty_struct *tty)
{
    int nr;
    char c;

    nr = CHARS(tty->write_q);
    while (nr--)
    {
        GETCH(tty->write_q, c);
        switch (state)
        {
        case 0:
            if (c > 31 && c < 127) //*在ascii中，这些就是可以显示的字符了
            {
                if (x >= video_num_columns)
                {
                    x -= video_num_columns;
                    pos -= video_size_row;
                    lf();
                }
                if (tty_hide == 1) //*如果需要hide，就调用
                {
                    c = '*';
                }
                __asm__("movb attr,%%ah\n\t"
                        "movw %%ax,%1\n\t" ::"a"(c),
                        "m"(*(short *)pos));
                pos += 2;
                x++;
            }
```

这个地方我设置的状态叫tty_hide，当其等于1的时候就终端全部输出为*，这样就实现了目标。

怎么切换状态

这就是个关键问题了，怎么切换系统状态，这个地方我使用的方法是在keyboard.S汇编程序中，当识别到F12按下，就切换状态。

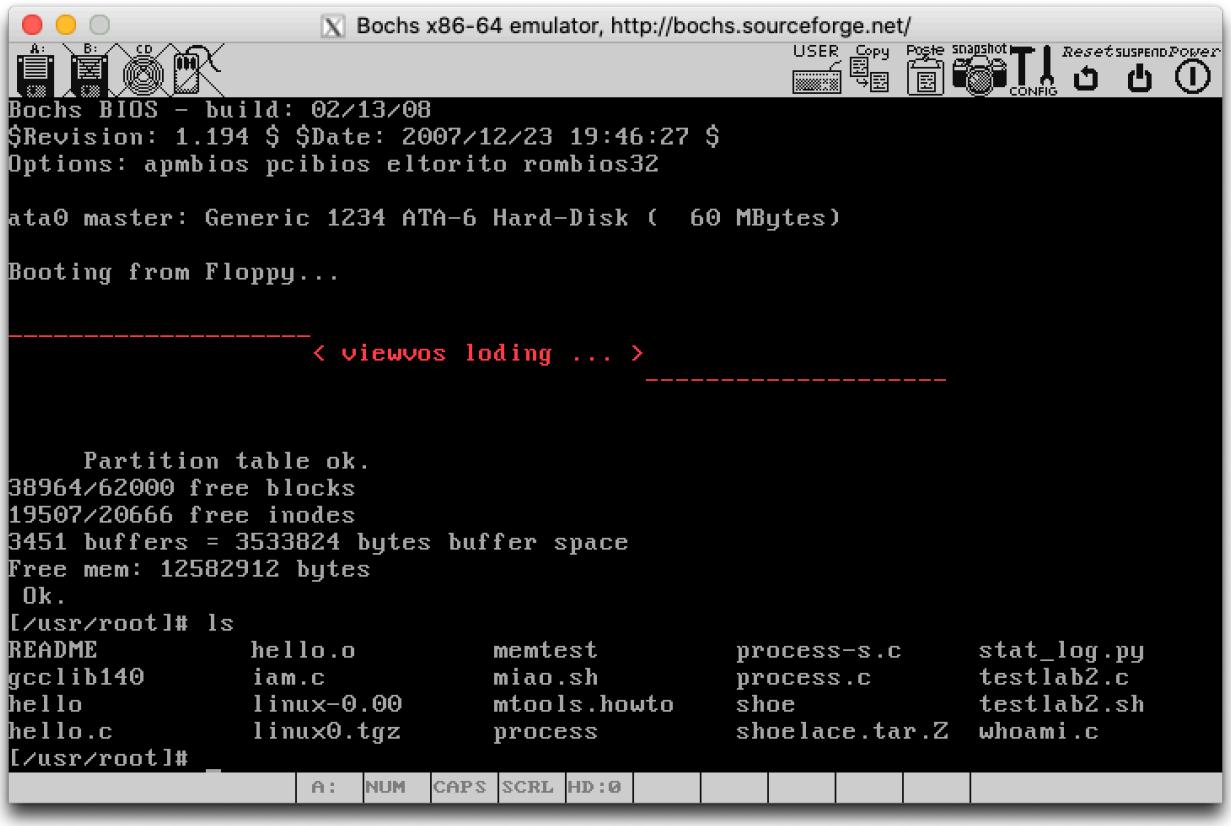
首先在sched.c中写好函数：

```c
void tty_hide_init(void)
{
    tty_hide = 0;
}
void change_tty_hide(void)
{
    tty_hide = !tty_hide;
    //printk("change tty_hide: %d\n", tty_hide);
}
```

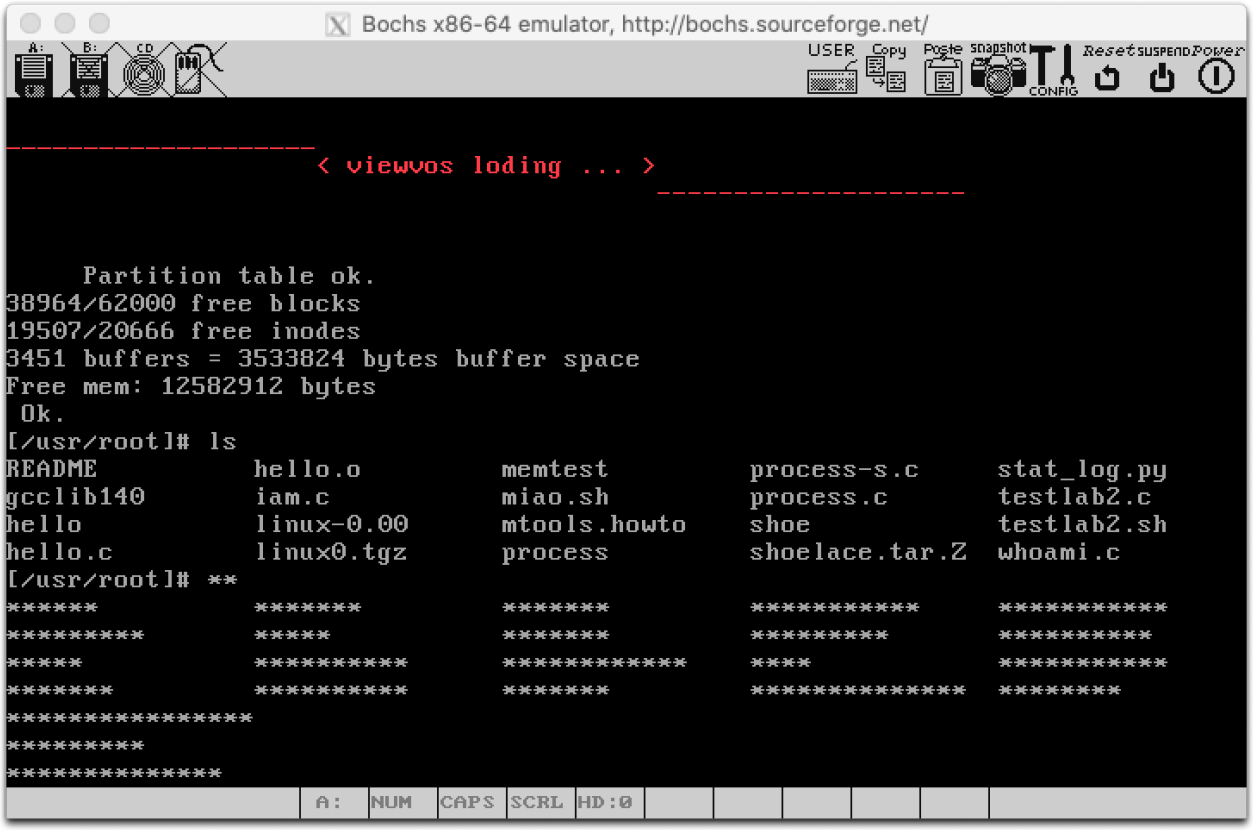
在sched.h头文件中定义好tty_hide变量，用于确定状态，之后在汇编程序中只要确认到按下F12，就调用change_tty_hide转换状态，至于为什么这里定义了一个init，我在看了这些代码的风格之后，就决定在系统启动也就是main函数中调用这个初始化状态，看着写法和Linus当时哪些写法一样的，这样就实现了状态切换。

运行结果

首先是刚开机没有按下F12的时候，这个时候不隐藏输出



之后按下F12，切换状态，就会隐藏都变成*



之后按下F12，切换回去，这样就又重新显示正常的字符。