# 操作系统原理实验报告

## 实验十一：多终端

数据科学与计算机学院　17大数据与人工智能

17341015　陈鸿峥

## 一、 实验目的

- 学习设备管理的方法，掌握其键盘和显示的驱动
- 扩展 MyOS 的内核，利用虚拟键盘和显示器的方法，实现进程的输入和输出的独立，即多终端

## 二、 实验要求

1. 禁止使用 BIOS 的键盘中断和屏幕输出调用，在 MyOS 实现自己的键盘输入驱动和显示器驱动。
2. 修改和扩展 MyOS 的系统调用，实现在 C 语言中，程序格式化输入和输出在独立的终端上进行，利用`CTRL+F1`、`CTRL+F2`、`CTRL+F3`可以切换不同的终端。

## 三、 实验环境

具体环境选择原因已在实验一报告中说明。

- Windows 10系统 + Ubuntu 18.04(LTS)子系统
- gcc 7.3.0 + nasm 2.13.02 + GNU ld (Binutils) 2.3.0
- GNU Make 4.1
- Oracle VM VirtualBox 6.0.6
- Bochs 2.6.9
- Sublime Text 3 + Visual Studio Code 1.33.1

虚拟机配置：内存4M，1.44M虚拟软盘引导，1.44M虚拟硬盘。

## 四、 实验方案

**本次实验继续沿用实验六保护模式的操作系统。用户程序都以ELF文件格式读入，并且在用户态(ring 3)下执行。**

多终端看上去内容很少，但实际上要修改的内容还是挺多的，涉及到进程和文件系统的处理。

**1. 多终端保护**

在terminal.h中创建终端的结构体，将**显示器、光标、文件路径**全部记录下来。同时创建一个terminal_list统一进行管理。

```
typedef struct terminal
{
    int num;
    int cursor;
    char buf[VIDEO_SIZE];
    char path[50];
    int dir;

} terminal_t;
static terminal_t terminal_list[MAX_TERMINAL];
```

还需要在进程中添加一个terminal项，用来管理进程是在哪个终端创建的，则该进程只会在那个终端输出。

为辨别不同的切换情况，这里我采用了两个指针，如下

```
terminal_t *curr_terminal, *tmp_terminal;
```

前一个指针记录当前终端，也就是显示的终端；后一个指针记录当前进程对应的中断，即该进程要输出的终端号。当切换进程时，切换的是tmp_terminal，而键盘切换的则是curr_terminal。

而输出到哪里也主要依据这两个终端号是否相同，通过下列函数获取对应的输出地址。

```
uintptr_t get_terminal_buf_addr()
{
    if (curr_terminal->num == tmp_terminal->num)
        return VIDEO_ADDRESS;
    else
        return (uintptr_t)&tmp_terminal->buf;
}
```

实际上终端的切换跟进程的切换是类似的，需要保护现场、恢复现场。

```
void change_terminal(int new_ter)
{
    disable();
    memcpy((void*)curr_terminal->buf,(void*)VIDEO_ADDRESS,VIDEO_SIZE);
    curr_terminal->cursor = get_cursor();
    get_curr_path(curr_terminal->path);
    curr_terminal->dir = get_curr_dir();
    if (get_proc() == NULL)
        curr_terminal = tmp_terminal = &terminal_list[new_ter];
```

```
    else
        curr_terminal = &terminal_list[new_ter];
    clear_screen();
    memcpy((void*)VIDEO_ADDRESS,(void*)curr_terminal->buf,VIDEO_SIZE);
    set_cursor(curr_terminal->cursor);
    set_curr_path(curr_terminal->path);
    set_curr_dir(curr_terminal->dir);
    enable();
}
```

**2. 键盘中断**

由于在实验六时进入保护模式，键盘中断已经是自己写的，故只需在原有基础上对功能键进行判断即可。

```
void keyboard_handler_main(void)
{
    unsigned char status;
    unsigned char scancode;

    /* write EOI */
    port_byte_out(0x20, 0x20);

    status = port_byte_in(KEYBOARD_STATUS_PORT);
    /* Lowest bit of status will be set if buffer is not empty */
    if (status & 0x01) {
        scancode = port_byte_in(KEYBOARD_DATA_PORT);
        if (scancode < 0)
            return;
        if (scancode & 0x80) {
            if (scancode - 0x80 == KEY_CTRL)
                __ctrl = false;
        } else {
            if (scancode == KEY_CTRL)
                __ctrl = true;
            else if (__ctrl && scancode == KEY_F1)
                change_terminal(0);
            else if (__ctrl && scancode == KEY_F2)
                change_terminal(1);
            else if (__ctrl && scancode == KEY_F3)
                change_terminal(2);
            else {
                char ascii = asccode[(unsigned char) scancode][0];
                kb_char = ascii;
            }
        }
```

```
    }
}
```

## 五、 实验结果

多终端的效果如图1和图2所示，由CHZOS后的标号可以看见是几号终端。

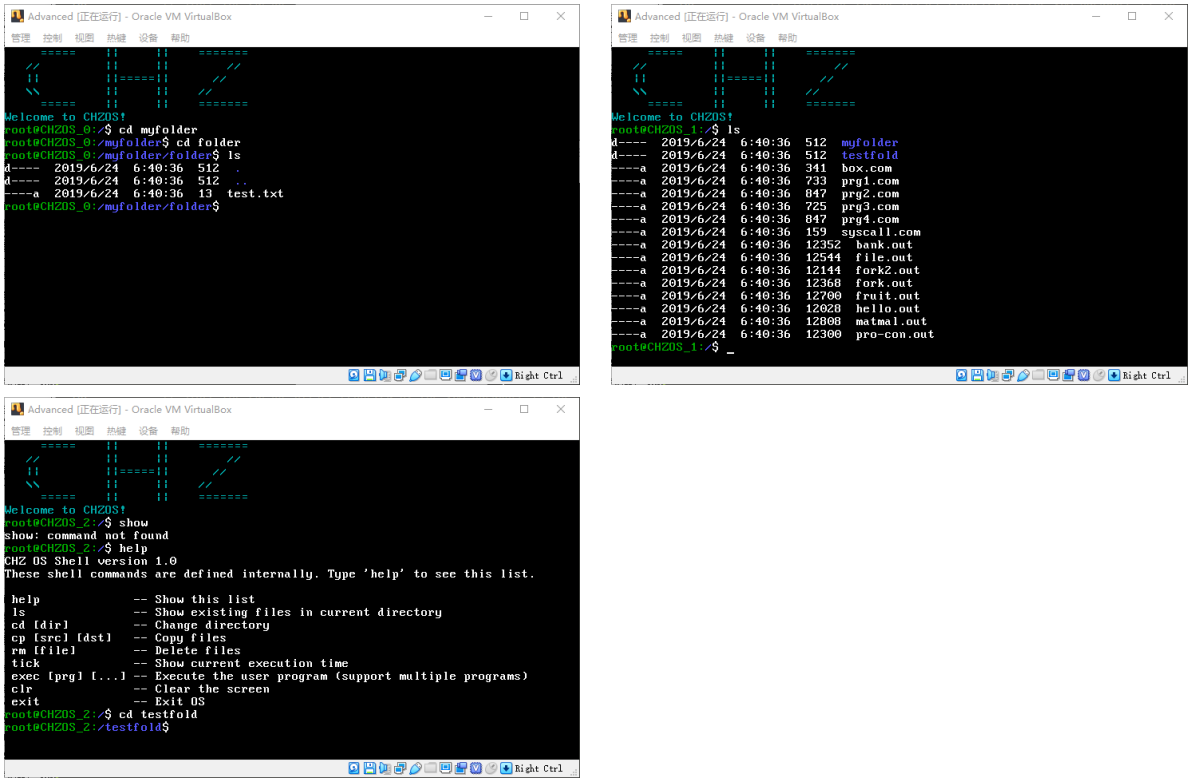


图 1: 三个终端初始状态，注意它们分别处在不同的目录。由于保护了目录信息，故切换终端不会导致目录出错。

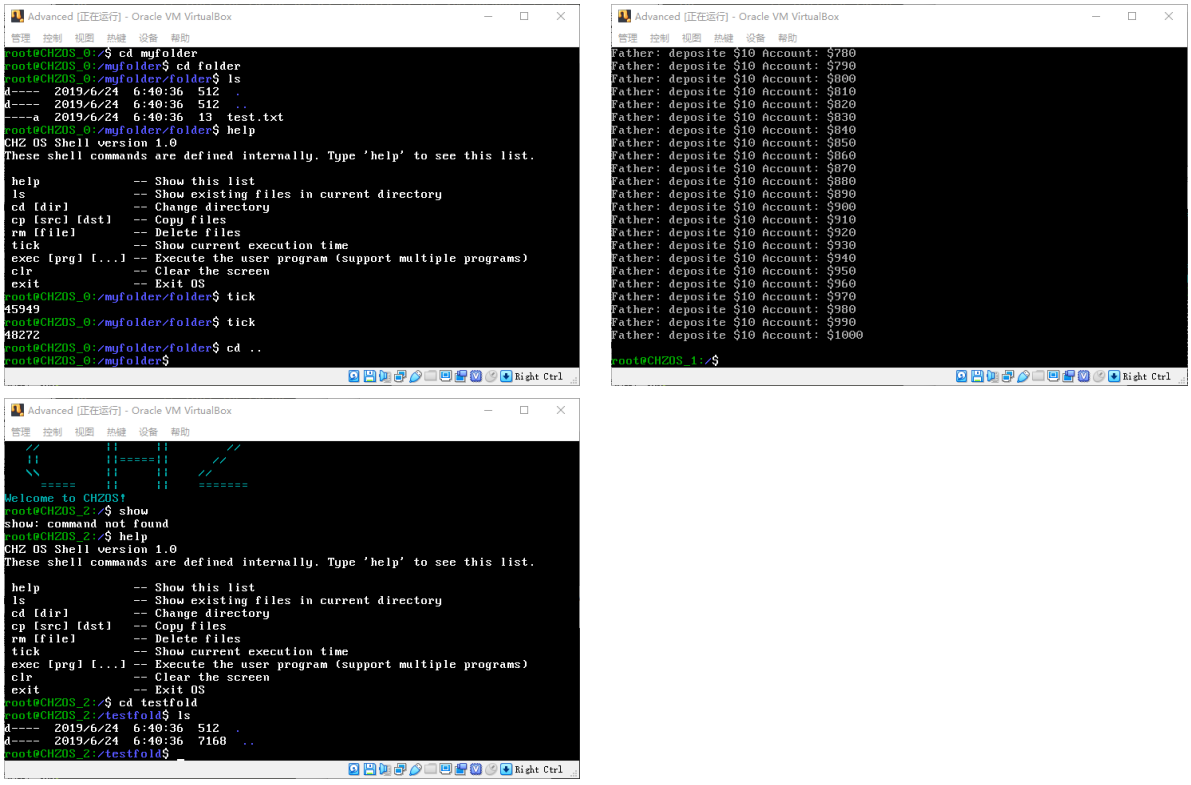


图 2: 在第二个终端运行进程后，再在其他终端进行操作（如执行其他进程、展示文件列表、计时等），发现操作系统能够正常运行，并且不会产生错误。

## 六、 实验总结

本次实验是操作系统的最后一次实验，虽然不是很难，但依然要修改的东西比较多。由于没有可以参考的代码，故所有的实现方法都是自己想的。最终想到用两个不同的指针实现多终端的切换，并实施出来验证了自己的想法，还是挺开心的。一开始没考虑周全，只考虑了进程切换的情况，经过调试发现文件系统也会与多终端有关，故要对文件系统也提供多终端支持，否则更换终端之后文件目录还没有换，这会导致文件索引时出现大问题。

总的来说，一个学期将操作系统11个实验完整实施下来，学到了非常非常多的东西。老师上课一直调侃说我们总会说`I hate OS!`，但于我而言，操作系统的实验真是一个**痛并快乐着**的过程。这一个学期下来不知上网搜索了多少资料，阅读了多少英文文献。看过几千页的Intel CPU说明书，也看过人家真实十几万行的操作系统代码。光是看懂其实就有一定难度，毕竟资料并不一定全面，还要将其完整实施出来更是难上加难。我一直跟自己说，全世界只有你懂你自己的操作系统，所以遇到问题也只有我自己能够解决。就是这样，我跪着也要把操作系统给做完，因为只有我自己懂它。

这其中的收获是巨大的，一方面对操作系统理论有了十分透彻的理解，知道这些理论到底怎么指导实际的实施；另一方面则是对各种工具的使用更加灵活，如C和x86汇编的编写，Makefile及Windows

10子系统WSL的使用等等。其实顺带着也把编译原理也给稍微学了一下，因为常常要看C语言编写的程序转换成汇编的结果，还要理解诸如函数堆栈等调用法则，这些其实都跟编译密切相关。

我也跟信计的同学讨论过，他们使用UCore进行实验，到底是比我们更难还是更简单。经过一个学期的实践，我确信我们的方式代码量更大，学到的东西也更多。我看过UCore的代码，基本的函数都已经封装好了，学生只用编写对应实验中几个函数即可，这其实与算法设计又有什么区别呢？但操作系统不是算法设计，它是一个**系统**，如何做好封装、设计简单易用的接口，如何将各个部分有机地耦合在一起，如何组合才能使性能达到最优，这些问题都是系统设计者需要思考的，而仅仅是函数的编写相当于把这些部分都抹除了，更加细节的东西不清楚，而只知大概不知全貌。因此，我特别感谢老师能给我们一次从下到上的操作系统游览之旅，这一趟下来真是获益匪浅。

图3清晰展示了我这学期肝保护模式操作系统的状况，中间有段时间在忙其他科目的大作业，因此就停滞了一段时间。待到期末才终于挤出点时间来，可以好好将我的操作系统彻底完善。最终新增代码量破万，着实算是一个中型的项目了。

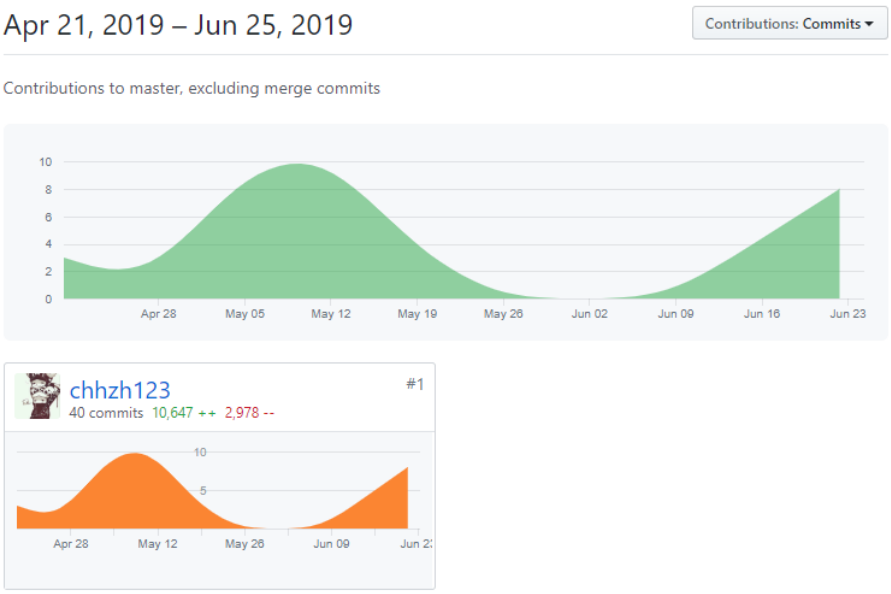


图 3: Git提交记录

最后的最后，为本学期操作系统实验完结撒花，希望我以后还能保持这样的热情去挑战更为困难的难关!

## 七、 参考资料

**操作系统实现完整资料：**

1. OS Development Series, http://www.brokenthorn.com/Resources/OSDevIndex.html

2. Roll your own toy UNIX-clone OS, http://www.jamesmolloy.co.uk/tutorial_html/
3. The little book about OS development, http://littleosbook.github.io/
4. Writing a Simple Operating System from Scratch, http://www.cs.bham.ac.uk/~exr/lectures/opsys/10_11/lectures/os-dev.pdf
5. Intel® 64 and IA-32 Architectures Software Developer's Manual
6. UCore OS Lab, https://github.com/chyyuu/ucore_os_lab
7. CMU CS 15-410, Operating System Design and Implementation, https://www.cs.cmu.edu/~410/
8. 李忠，王晓波，余洁，《x86汇编语言-从实模式到保护模式》，电子工业出版社，2013

## 附录 A. 程序清单

### 1. 内核核心代码

| 序号 | 文件 | 描述 |
|---|---|---|
| 1 | bootloader.asm | 主引导程序 |
| 2 | kernel_entry.asm | 内核汇编入口程序 |
| 3 | kernel.c | 内核C入口程序 |
| 4 | Makefile | 自动编译指令文件 |
| 5 | bootflpy.img | 引导程序/内核软盘 |
| 6 | mydisk.hdd | 虚拟硬盘 |
| 7 | bochsrc.bxrc | Bochs配置文件 |

**2. 内核头文件**

| 序号 | 文件 | 描述 |
|---|---|---|
| 1 | disk_load.inc | BIOS读取磁盘 |
| 2 | show.inc | 常用汇编字符显示 |
| 3 | gdt.inc | 汇编全局描述符表 |
| 4 | gdt.h | C全局描述符表 |
| 5 | idt.h | 中断描述符表 |
| 6 | hal.h | 硬件抽象层 |
| 6.1 | pic.h | 可编程中断控制器 |
| 6.2 | pit.h | 可编程区间计时器 |
| 6.3 | keyboard.h | 键盘处理 |
| 6.4 | tss.h | 任务状态段 |
| 6.5 | ide.h | 硬盘读取 |
| 7 | io.h | I/O编程 |
| 8 | exception.h | 异常处理 |
| 9 | syscall.h | 系统调用 |
| 10 | task.h | 多进程设施 |
| 11 | user.h | 用户程序处理 |
| 12 | terminal.h | Shell |
| 13 | scancode.h | 扫描码 |
| 14 | stdio.h | 标准输入输出 |
| 15 | string.h | 字符串处理 |
| 16 | elf.h | ELF文件处理 |
| 17 | api.h | 进程管理API |
| 18 | semaphore.h | 信号量机制 |
| 19 | systhread.h | 线程模型 |
| 20 | pthread.h | 线程管理API |
| 21 | fat12.h | FAT12文件系统 |
| 22 | sysfile.h | 虚拟文件系统 |
| 23 | file.h | 文件管理API |

**3. 用户程序**

用户程序都放置在usr文件夹中。

| 序号 | 文件 | 描述 |
|---|---|---|
| 1-4 | prgX.asm | 飞翔字符用户程序 |
| 5 | box.asm | 画框用户程序 |
| 6 | sys_test.asm | 系统中断测试 |
| 7 | fork_test.c | 进程分支测试 |
| 8 | fork2.c | 进程多分支测试 |
| 9 | bank.c | 银行存取款测试 |
| 10 | fruit.c | 父子祝福水果测试 |
| 11 | prod_cons.c | 消费者生产者模型测试 |
| 12 | hello_world_thread.c | 多线程Hello_world测试 |
| 13 | matmul.c | 多线程矩阵乘法测试 |
| 14 | file.c | 文件读写测试 |

## 附录 B. 系统调用清单

| int 0x80功能号 | 功能 |
|---|---|
| 0 | 输出OS Logo |
| 1 | 睡眠100ms |
| 10 | fork |
| 11 | wait |
| 12 | exit |
| 13 | get_pid |
| 20 | get_sem |
| 21 | sem_wait |
| 22 | sem_signal |
| 23 | free_sem |
| 100 | 返回内核Shell |

| int 0x81功能号 | 功能 | int 0x82功能号 | 功能 |
|---|---|---|---|
| 0 | pthread_create | 0 | fopen |
| 1 | pthread_join | 1 | fclose |
| 2 | pthread_self | 2 | fread |
| 3 | pthread_exit | 3 | fwrite |
| | | 4 | fgets |
| | | 5 | fputs |