目录

目录
一、实验目的
二、实验要求
三、实验环境
四、实验方案
（一）、设计整体架构
（二）、具体实现
1、分离引导器和内核
2、g++与nasm混合编程
(1)nasm调用C++函数
(2)C++调用nasm函数
(3)编译链接
3. 内核模块实现
(1)直接操作硬件的内核函数
(2)TTY模块
(3)用户程序加载器(bin_loader)模块
4、C语言库的实现
(1)系统调用实现
(2)库函数的实现
(5) 交互终端（shell）的实现
功能说明：
实现说明：
(6) 实现中缀表达式计算器
四、实验过程和结果
图一：操作系统启动后的界面
图二：显示所有用户程序
图三：批处理演示（执行了两个回显和一个计算器程序）
图四：历史记录功能
图五：自动补全功能（输入cl后按下tab键）
图五：调用计算器程序执行混合四则运算运算
图六：调用实验二的贪吃蛇程序
图七：显示帮助信息
五、实验总结
六、参考文献

一、实验目的

1. 把原来在引导扇区中实现的监控程序(内核)分离成一个独立的执行体，存放在其它扇区中，为“后来“扩展内核提供发展空间。
2. 学习汇编与c混合编程技术，改写实验二的监控程序，扩展其命令处理能力，增加实现实验要求2中的部分或全部功能。

二、实验要求

- 实验三必须在实验二基础上进行，保留或扩展原有功能，实现部分新增功能。
- 监控程序以独立的可执行程序实现，并由引导程序加载进内存适当位星，内核获得控制权后开始显示必要的操作提示信息，实现若干命令，方便使用者(测试者)操作。
- 制作包含引导程序，监控程序和若干可加载并执行的用户程序组成的1.44M软盘映像。

三、实验环境

本次实验使用了汇编与**C++**混合编程，因此新增了以下工具

C++编译器：g++ 7.3.0，Target: i386-elf

链接器：ld 2.30

二进制文件分析器：objdump 2.30

符号分析器：nm 2.3.0

使用C++而非C语言的原因有：

1. 对于实现同样功能的代码，C++和C语言产生的汇编指令是基本相同的，都可以在裸机上正常执行。C++对C语言有很强的兼容性，同时由于C++标准模板库依赖内存管理机制，本次实验中我**实现的是C语言标准库（printf，scanf，strcpy等），因此实验报告中的技术细节基本同样适用于C语言**。
2. C++相对于C在语法层面添加了bool类型、传递引用、using语句（代替typedef）、template模板、auto自动类型推导等语法规则，更加高效和方便。我们大一时也是从一上来就从C++学起，所以很习惯使用C++。
3. C++是面向对象编程的，易于实现内核功能的模块化。
4. C++支持命名空间机制，通过这个机制我可以把同一份代码不加修改，只调整编译参数，就能移植到在物理机上进行调试。使用了Google Test单元测试框架。

其余环境与之前实验大致相同：

主机操作系统：Mac OS 10.12

编辑器：Vim 8.0.1400、VS Code 1.21.0

汇编器：Nasm 2.13.02

虚拟机、调试器：Bochs 2.6.9

版本控制：Git 2.15.1

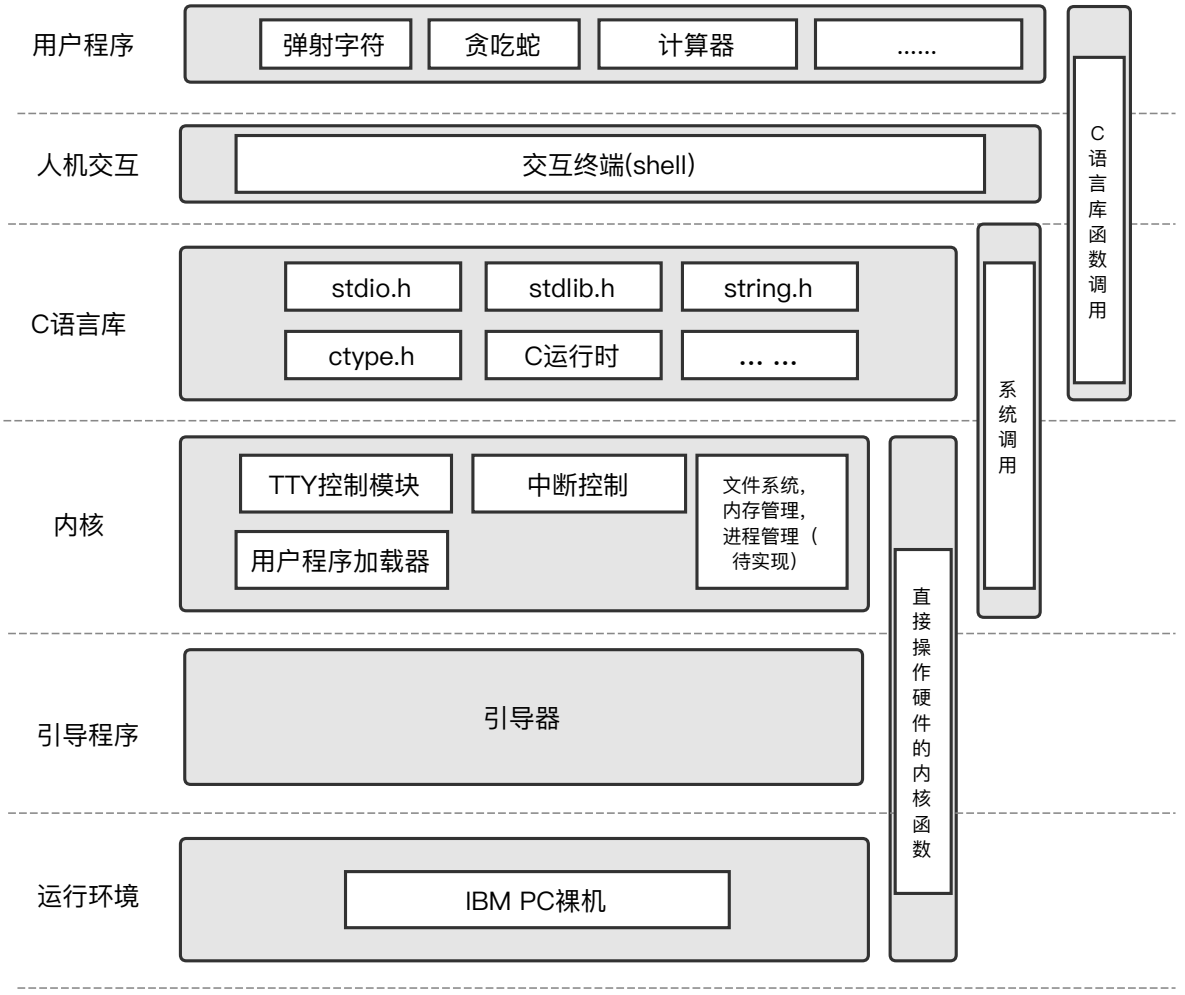
自动构建：GNU Make 3.8.1

四、实验方案

（一）、设计整体架构

操作系统作为一个复杂的系统工程，只有从软件工程的角度首先建立好一个良好的框架，才能实现系统的高内聚、低耦合，确保可调试、可扩展性。

本次实验中我构建好了本操作系统的基本框架，如下图所示，上层基于下层，最右边两个跨层的竖条表示上层用到的，下层为之层提供的服务。



具体的文件如下：

```
.
├── Makefile                    //主Makefile
├── basic_lib                   //内核函数库
|   ├── Makefrag                //该模块的makefile
|   ├── asm_lib.asm             //汇编语言编写的部分
|   ├── sys_io.cpp              //C++编写的部分
|   ├── sys_lib.h               //头文件
├── bochsrc                     //bochs启动脚本
├── boot                        //引导器模块
|   ├── Makefrag
|   └── bootloader.asm          //引导程序
├── gdbdash                     //gdb相关，下同
├── gdbinit.part
├── include                     //公共头
|   └── defines.h
```

```
├── kernel                          //内核
|   ├── Makefrag
|   ├── bin_loader.h                //用户程序加载器模块
|   ├── kernel_main.cpp             //内核C++部分
|   ├── kernel_start.S              //内核汇编部分
|   ├── sh.h                        //shell模块
|   └── tty.h                       //tty模块
├── libc                            //C函数库
|   ├── Makefile
|   ├── Makefrag
|   ├── cstart.S
|   ├── ctype.cpp
|   ├── ctype.h
|   ├── stdio.cpp
|   ├── stdio.h
|   ├── stdlib.cpp
|   ├── stdlib.h
|   ├── string.cpp
|   ├── string.h
|   ├── sys
|   |   └── hhos.h                  //系统调用库
|   ├── test.cpp                    //测试程序
├── linker.ld                       //链接选项文件
├── mkinc                           //Makefile库
|   ├── __gmsl
|   └── gmsl
├── usr                             //用户程序
|   ├── Makefrag
|   ├── bc.cpp                      //计算器程序
|   ├── common.asm                  //弹射字符程序公共部分
|   ├── help.asm                    //操作系统帮助文件
|   ├── linker.ld                   //用户C++程序链接选项文件
|   ├── record.asm                  //用户程序记录文件
|   ├── user1.asm                   //四个弹射字符程序
|   ├── user2.asm
|   ├── user3.asm
|   ├── user4.asm
|   └── user5.asm                   //贪吃蛇程序
```

（二）、具体实现

1、分离引导器和内核

将实验二的源代码中，除了读取软盘和跳转到内核的代码全部分离到单独的kernel_start.asm中，内核加载和启动地址设置为7e00h。

2、g++与nasm混合编程

(1)nasm调用C++函数

- C++源代码中写入内联汇编指令.code16gcc以生成16位机器码
- 函数声明：C++源文件中将函数声明为extern "C"（采用C语言链接时），nasm源文件中声明extern <C++函数名>。
- 函数调用：由于生成16位代码时，NASM默认在call时压栈两个字节，而g++在生成16位代码时依旧使用的是32位地址，因此如果要返回汇编，必须使用call dword <C++函数名>。
- 函数传参：nasm中使用push指令传入，和函数调用同样的原因，push参数时一定要push 32位的寄存器

(2)C++调用nasm函数

- 函数声明：nasm源文件中声明global <nasm函数名>
- nasm函数编写：nasm导出到C++的函数一定要以push bp开始（保存堆栈指针），结尾处pop bp。

  g++压入的返回地址是32位的，加上压入的bp的16位，可以算出第一个参数在bp+6处。压入的每个参数也是32位的，因此之后的参数依次在 bp + 10, bp + 14 .....
- 程序返回：注意一定要 pop ecx 然后 jmp cx，不然会出现两字节的栈内存泄露
- 返回值：nasm将返回值压入ax寄存器，C++中即可读取

(3)编译链接

Nasm与gcc各种汇编生成目标文件，再使用ld链接生成binary文件。

本次实验中继续扩展了Makefile/Makefrag，总长度达到了243行。能够全自动完成整个系统的构建和虚拟机的运行。

3. 内核模块实现

(1)直接操作硬件的内核函数

这些函数均以sys开头，提供了最底层的功能，如输入（BIOS中断）、输出（直接操作显存）、操作端口（使用内联汇编封装inb、outb）、读取软盘、执行用户程序。仅有内核可以调用这些函数。

函数

void	**sys_execve_bin** ()
void	**sys_dbg_bochs_putc** (char c)
void	**sys_bios_print_string** (const char *str, unsigned int len, int color, int pos)
void	**sys_bios_print_int** (int num, int color, int pos)
void	**sys_bios_clear_screen** ()
void	**sys_bios_putchar** (char c, int color, int x, int y)
int	**sys_bios_getchar** ()
uint8_t	**sys_inb** (uint16_t port)
void	**sys_outb** (uint16_t port, uint8_t data)
char	**sys_get_scancode** ()
int	**sys_getchar** ()
void	**sys_putchar** (int c, int color, int x, int y)
void	**sys_print_string** (const char *str, unsigned int len, int x, int y)
void	**sys_print_int** (int num, int x, int y)
void	**sys_read_disk** (uint32_t segment, uint32_t address, uint16_t logical_start_sector, uint8_t secotr_cnt)
void	**sys_bios_scroll_up** (int color)

- 读取软盘函数sys_read_disk()：支持将任意逻辑扇区号指定的软盘扇区加载入内存。这里用到了逻辑扇区号到物理柱面、磁头、扇区编号的转换。代码如下

```
#define FLOPPY_SECTOR_PER_TRACK 18
#define FLOPPY_TRACK_PER_HEAD 80
#define FLOPPY_HEAD_PER_DISK 2
uint8_t head = (logical_start_sector % (FLOPPY_HEAD_PER_DISK *
FLOPPY_SECTOR_PER_TRACK)) / FLOPPY_SECTOR_PER_TRACK;
uint16_t cylinder = logical_start_sector / (FLOPPY_HEAD_PER_DISK *
FLOPPY_SECTOR_PER_TRACK);
uint16_t sector = (logical_start_sector % (FLOPPY_HEAD_PER_DISK *
FLOPPY_SECTOR_PER_TRACK)) % FLOPPY_SECTOR_PER_TRACK + 1;
```

- 执行用户程序过程sys_execve_bin()：和实验二类似，首先设置PSP，然后跳转到用户程序代码，用户程序调用我实现的20h、21h号调用返回。

  与实验二不同的是，本次试验中我的内核地址空间（数据段）已经到了aec8h，因此我将用户程序加载到与内核不同的段中（1000:A100），因此必须手动修改ds、es、ss寄存器为1000h，然后再使用远程jmp，指定段地址1000。返回后再改回来。实现代码如下：

```
sys_execve_bin:
      push bp
      mov bp, sp
      pusha          ;这里要保护寄存器!!!
      push ds
      push es
      mov ax, 0x1000
      mov ds, ax
      mov es, ax
      mov ss, ax
      mov word[0xA000], 0xCD
      mov word[0xA000 + 2], 20h
      mov word[0xA00A], return_point
```

```
        mov word[0xA00A + 2], cs
        jmp 0x1000:0xA100
return_point:
        mov ax, 0x0000
        mov ss, ax
        pop ax
        mov es, ax
        pop ax
        mov ds, ax
        popa
        pop bp
        ret
```

(2)TTY模块

TTY(teletypewriter，电传打印机)是对80x25字符界面显示设备的抽象。管理了当前光标位置、提供了移动光标、滚屏、放置字符（能够正确解析回车、退格等转义字符）的功能。由于采用了面向对象设计，在未来实现了内存管理后，很容易实现多TTY功能。

Public 成员函数

	tty ()
void	**tty_init** ()
int	**get_x** ()
int	**get_y** ()
int	**get_color** ()
void	**set_x** (int x)
void	**set_y** (int y)
void	**set_color** (int _color)
void	**move_cursor** (int x, int y)
void	**scroll_up** ()
void	**putchar_worker** (int c, int **color**, int x, int y)
void	**putchar** (int c)

Private 属性

int	**cur_x**
int	**cur_y**
int	**color**

移动光标功能使用了操作端口的方法：

分别向0x3D4端口输出0x0F和0x0E选择15，14号显示器寄存器，然后0x3D5写光标的y/x坐标。

```
void move_cursor(int x, int y)
{
    uint16_t pos = x * 80 + y;
    sys_outb(0x3D4, 0x0F);
    sys_outb(0x3D5, (uint8_t) (pos & 0xFF));
    sys_outb(0x3D4, 0x0E);
    sys_outb(0x3D5, (uint8_t) ((pos >> 8) & 0xFF));
}
```

滚屏功能使用了10号BIOS调用的6号功能

```
void sys_bios_scroll_up(int color)
{
asm volatile              //使用内联汇编
("pusha\n\t"
"movb $1, %%al\n\t"       //上移一行
"movb %0, %%bh\n\t"
"movb $0, %%ch\n\t"       //范围设为整个屏幕
"movb $0, %%cl\n\t"
"movb $24, %%dh\n\t"
"movb $79, %%dl\n\t"
"movb $0x06, %%ah\n\t"
"int $0x10\n\t"
"popa\n\t"
:
: "g"(color));
}
void scroll_up()
{
    if (cur_x >= 25)
    {
        sys_bios_scroll_up(color);
        --cur_x;
    }
}
```

(3)用户程序加载器(bin_loader)模块

对外提供了load_binary_from_floppy函数，将放置在指定扇区的binary格式用户程序加载到1000:A100处。

静态 Public 成员函数

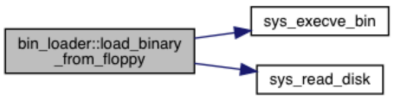
static void **load_binary_from_floppy** (int n)

静态 Private 属性

static constexpr **uint32_t** **user_prog_load_addr** = 0xA100

static constexpr **uint32_t** **user_prog_segment** = 0x1000

load_binary_from_floppy函数调用sys_execve_bin(), sys_read_disk()函数实现加载功能。



4、C语言库的实现

(1)系统调用实现

这部分在下一个实验报告中将详述。最后实现的效果是提供了98H号系统调用、并提供了`static int system_call_getchar ()`和`static void system_call_putchar (int ch)`这两个C语言接口。

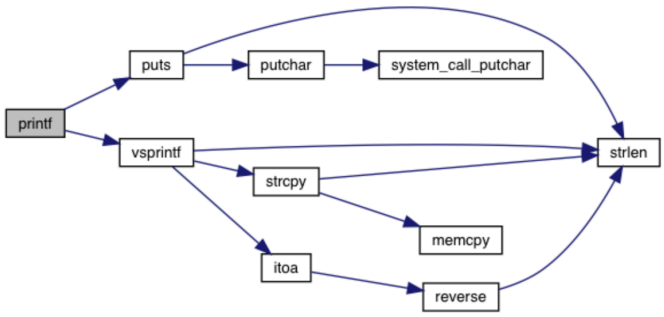
(2)库函数的实现

目前我实现了以下函数。这些函数的定义与C语言完全吻合。功能上也基本与标准兼容。

- stdio.h

```
int     putchar (int ch)
int     puts (const char *string)
int     vsprintf (char *buffer, const char *format, va_list vlist)
int     sprintf (char *buffer, const char *format,...)
int     printf (const char *format,...)
int     getchar (void)
char *  gets (char *str)
int     vsscanf (const char *buffer, const char *format, va_list vlist)
int     sscanf (const char *buffer, const char *format,...)
int     scanf (const char *format,...)
```

  printf和scanf目前支持%c %s %d %i %o %x %X %u 八种控制符。它们也是目前实现的库里面最复杂的函数。以printf为例，实现的方式是：首先调用vsprinf处理控制字符串和输入变量将要打印的内容写入一个printfbuf字符数组中，然后调用puts函数将printfbuf打印出来。puts函数最终是调用系统调用system_call_putchar的。



- stdlib.h

```
long    strtol (const char *str, char **str_end, int base)
```

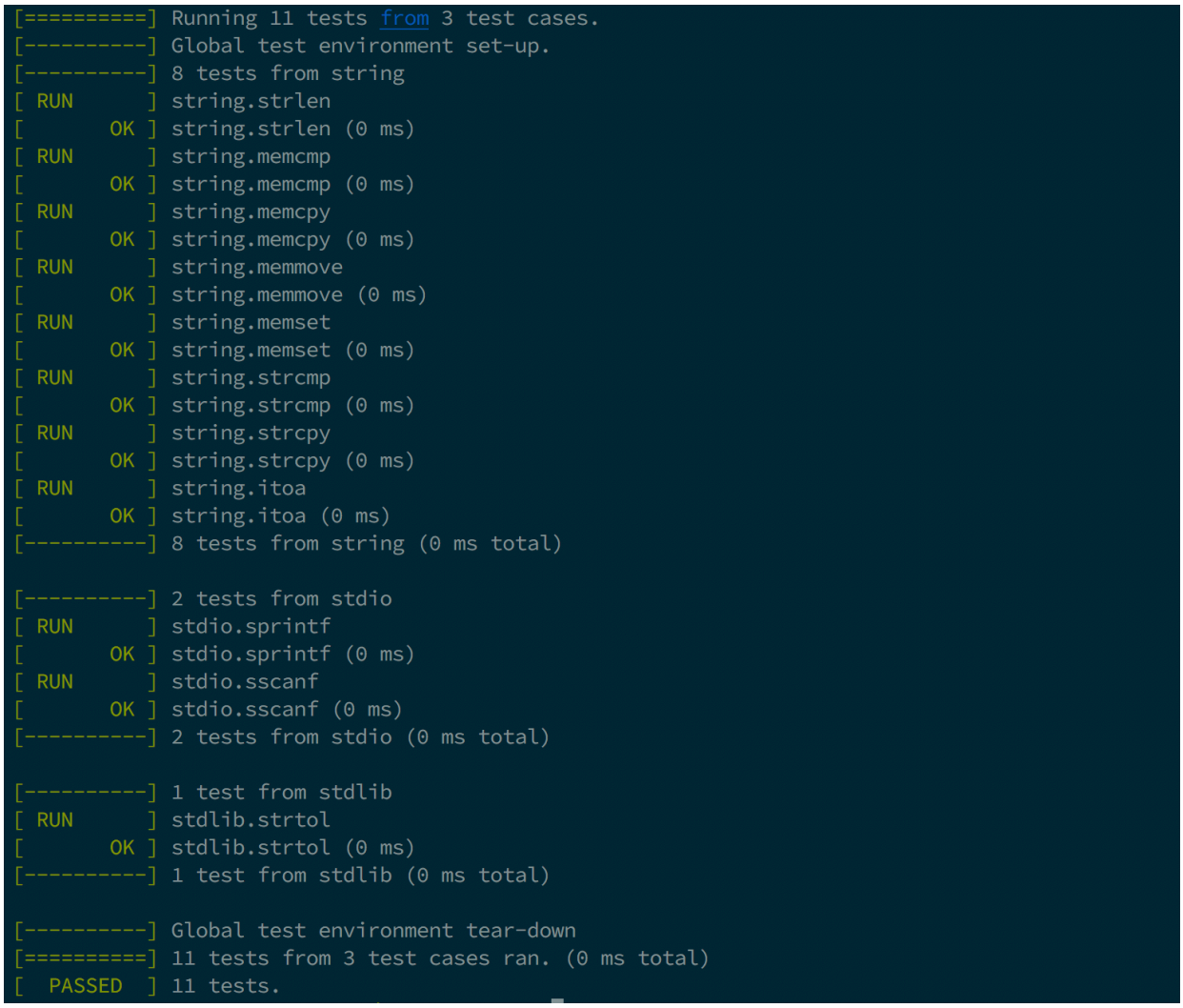
- string.h

```
int     memcmp (const void *_s1, const void *_s2, size_t n)
void *  memcpy (void *_dst, const void *_src, size_t n)
void *  memmove (void *_dst, const void *_src, size_t n)
void *  memset (void *_dst, int c, size_t n)
size_t  strlen (const char *_str)
char *  strcpy (char *_dst, const char *_src)
int     strcmp (const char *_s1, const char *_s2)
void    utoa (char *buffer, unsigned int num, int base)
void    reverse (char *buffer)
template<typename T >
void    itoa (char *buffer, T num, int base, bool captial=false)
```

- ctype.h

```
int     isspace (int ch)
int     isalnum (int ch)
int     isdigit (int ch)
```

库函数编写完后，我定义了一个_HHOS_LIBC_TEST宏，如果该宏开启，就会将这些自己实现的库函数隐藏到hhlic命名空间中，同时将我实现的系统调用替换为物理机的系统调用。通过这个方式，我可以在物理机上对这些函数进行正确性测试，以下是使用Google Test框架测试的结果，可见函数都通过了测试。



(5) 交互终端（shell）的实现

功能说明：

本shell支持以下特色功能：

- **自动补全命令**（按tab键）
- **批处理**（使用 ; 分割多个命令）
- **历史记录功能** （目前支持十条）
- 帮助文件和用户程序记录采用读取文件方式，而非硬编码

支持以下内置命令

- ls 或 dir：显示所有用户程序及所在的逻辑扇区号
- cls 或 clear：清屏
- echo：回显输入

实现说明：

shell由sh.h中的sh类实现，对外提供初始化功能的默认构造函数和run函数。

初始化时首先使用memset情况输入命令缓冲区，然后读取帮助文件和用户程序记录（每个保存在prog_entry类中)。

run函数使shell开始运行，进入输入->求值的死循环，对不同的按键做出处理，如退格，Tab（使用bf函数进行字符串对比，进行自动补全提示），普通按键（显示并放入缓冲区）以及回车。

按下回车后，首先会将目前的输入放进历史记录(history_push)，依据空格将输入拆分为单个的词(split_input)，然后依据 ; 判断有多少条命令(split_batch)。使用一个cmd类型的结构体记录这条命令开始的词和词数，传递给exec函数执行。exec函数通过is_command函数比对调用指定的命令或程序。最后程序结束返回清空输入缓冲区，重新显示提示符。

类

struct	**cmd**
struct	**prog_entry**

Public 成员函数

	sh ()
void	**run** ()

Private 成员函数

bool	**is_command** (const **cmd** &input_cmd, const char *cmd_name)
void	**history_push** (const char ***buf**)
int	**exec** (const **cmd** &input_cmd)
void	**read_prog_record** ()
void	**read_help_file** ()
int	**split_input** (char ***buf**)
int	**split_batch** (char ****inputs**, int input_cnt)
int	**bf** (const char *Pattern, const char *Text)

(6) 实现中缀表达式计算器

在完成了一些C函数库和用户程序加载器的情况下，我实现了第6个用户程序，一个支持混合四则运算和错误提示的计算器。本次实验中由于没有内存管理，没法实现动态的vector，因此实现的功能比较简单，操作数只能是个位的。但其意义在于验证了我的C函数库的可用性和加载C++程序并返回操作系统的可行性。

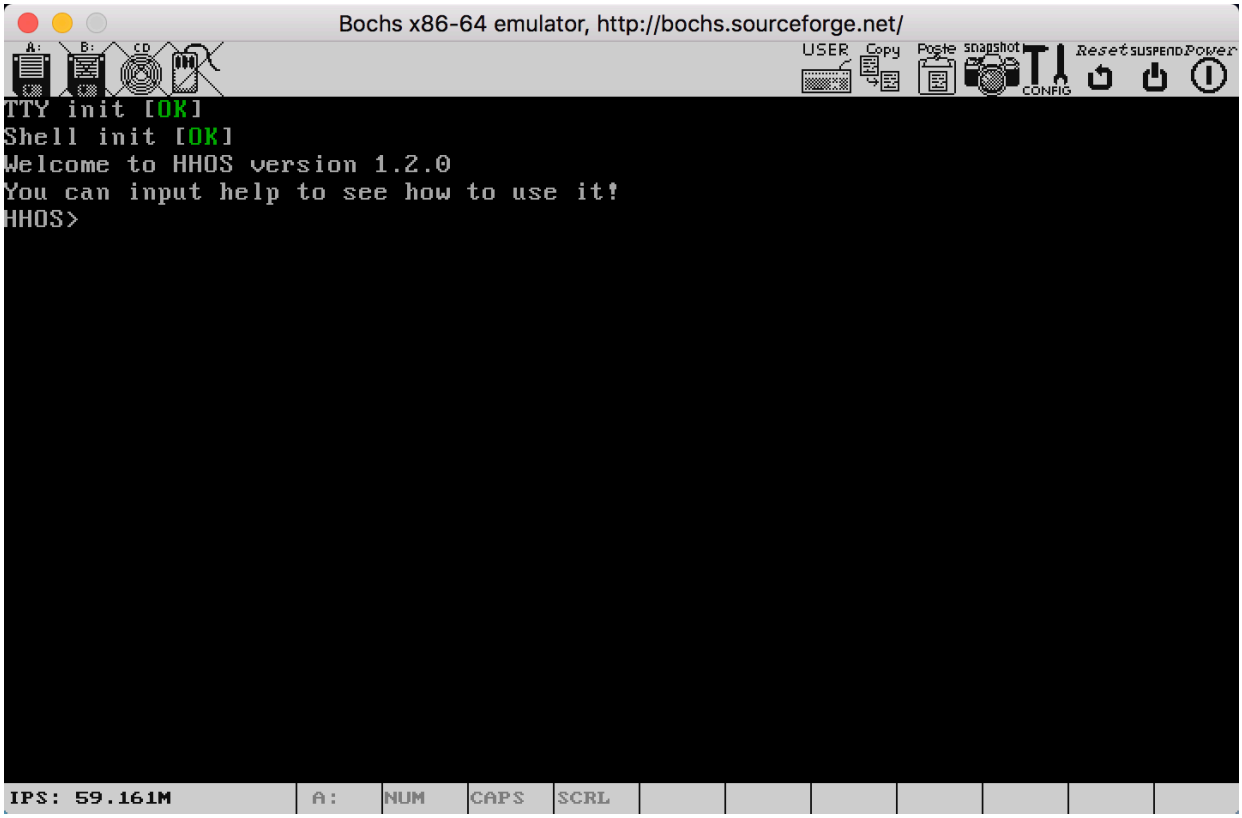
程序首先使用中缀表达式转后缀表达式算法将输入转为后缀表达式。然后使用后缀表达式求值得出结果。

遇到非法输入程序会提示错误的位置。

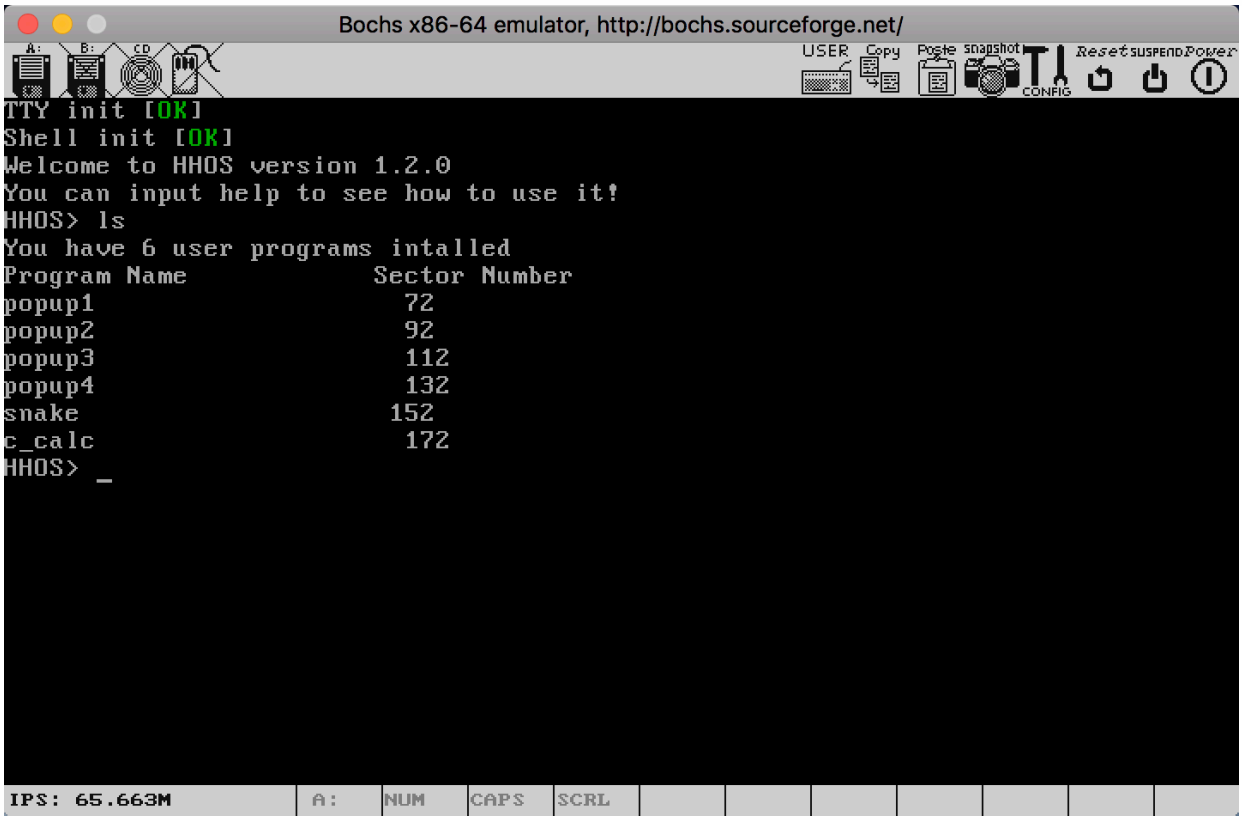
输入q退出计算器程序。

四、实验过程和结果

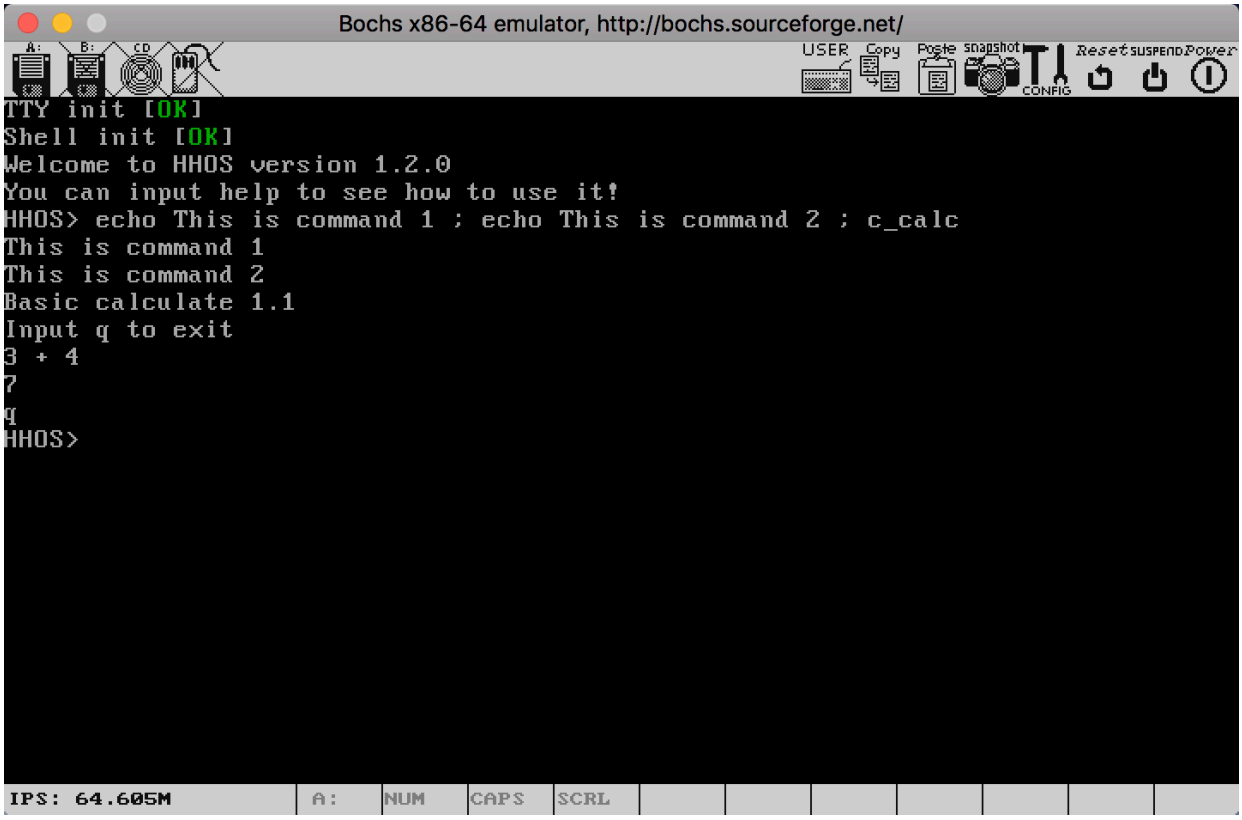
图一：操作系统启动后的界面



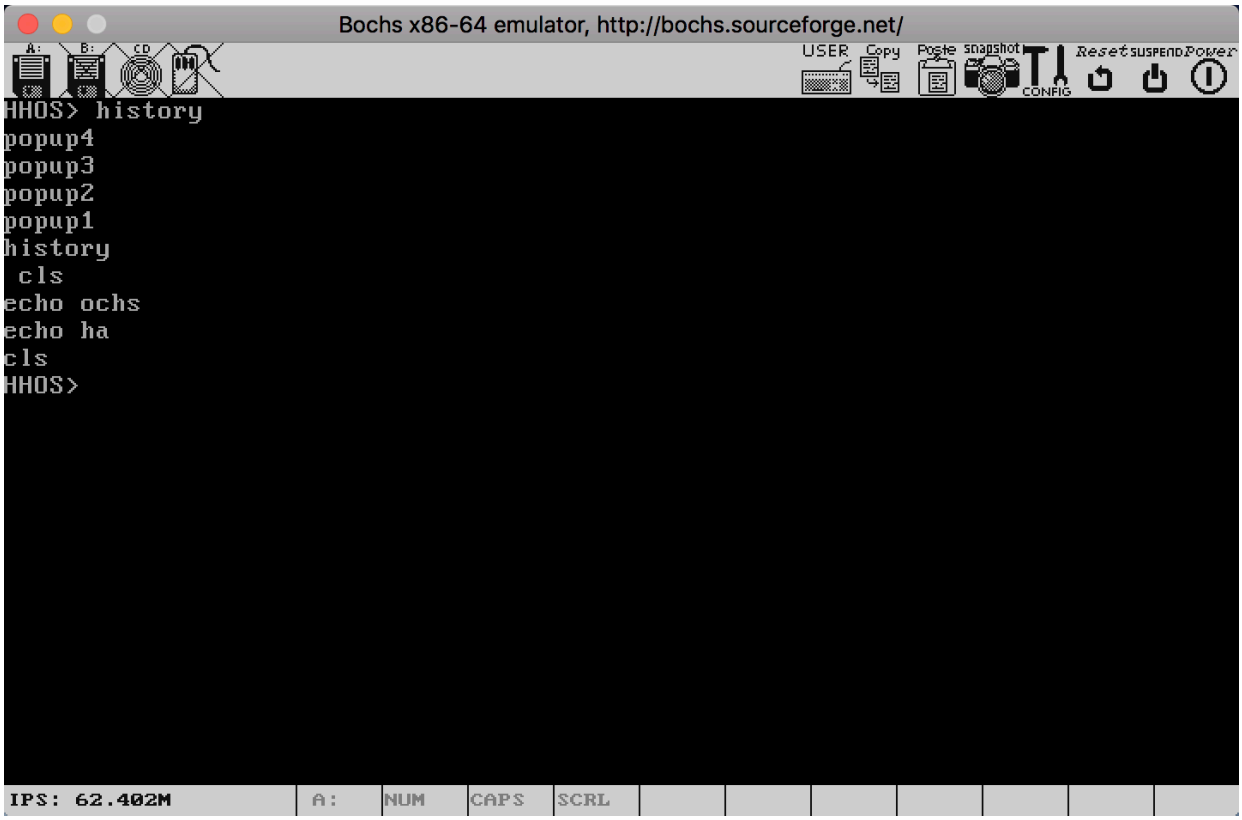
图二：显示所有用户程序



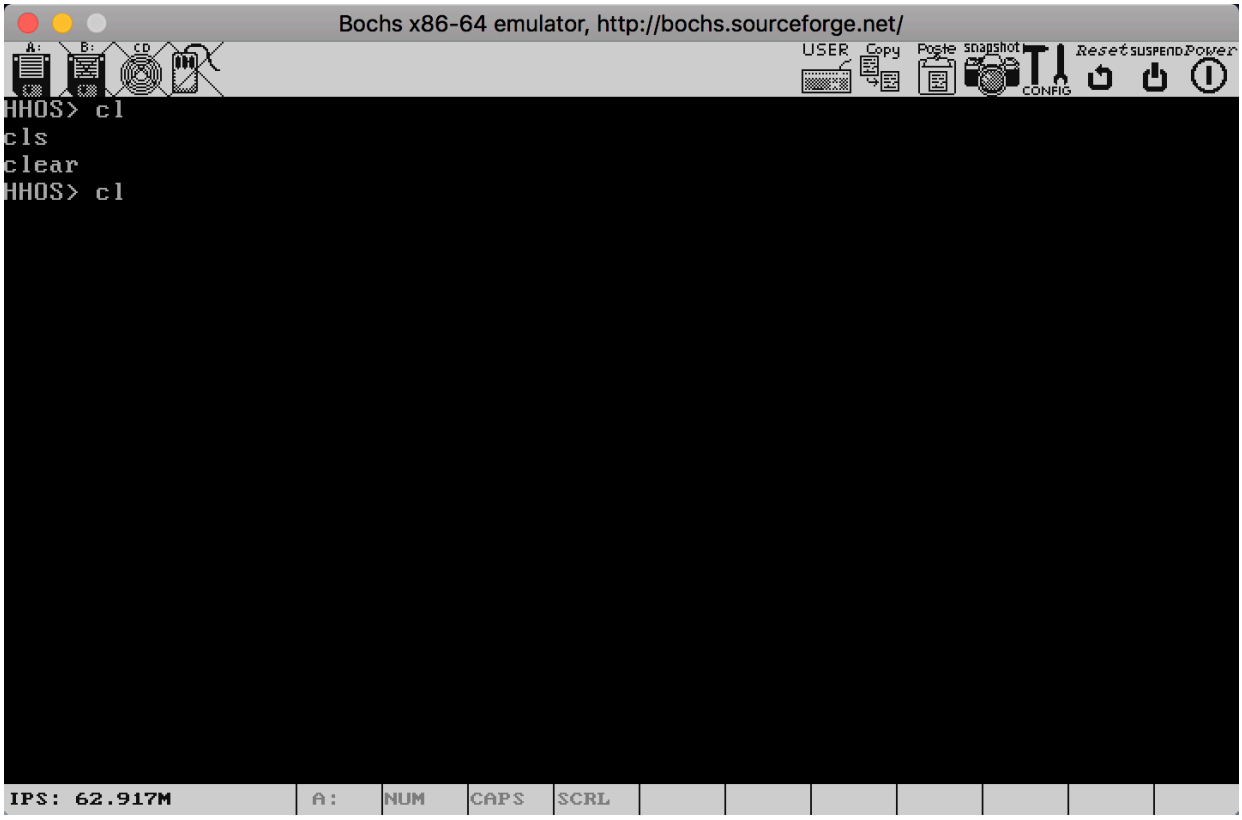
图三：批处理演示（执行了两个回显和一个计算器程序）



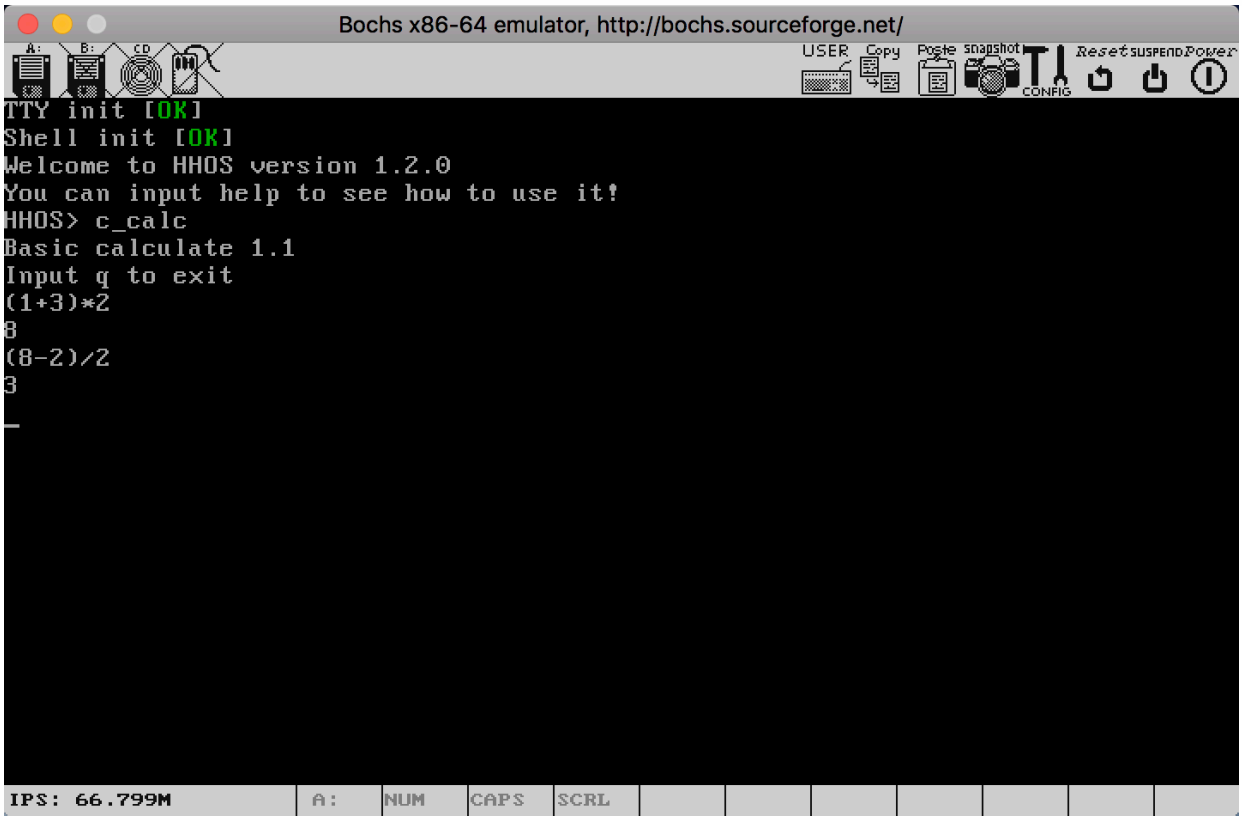
图四：历史记录功能



图五：自动补全功能（输入cl后按下tab键）

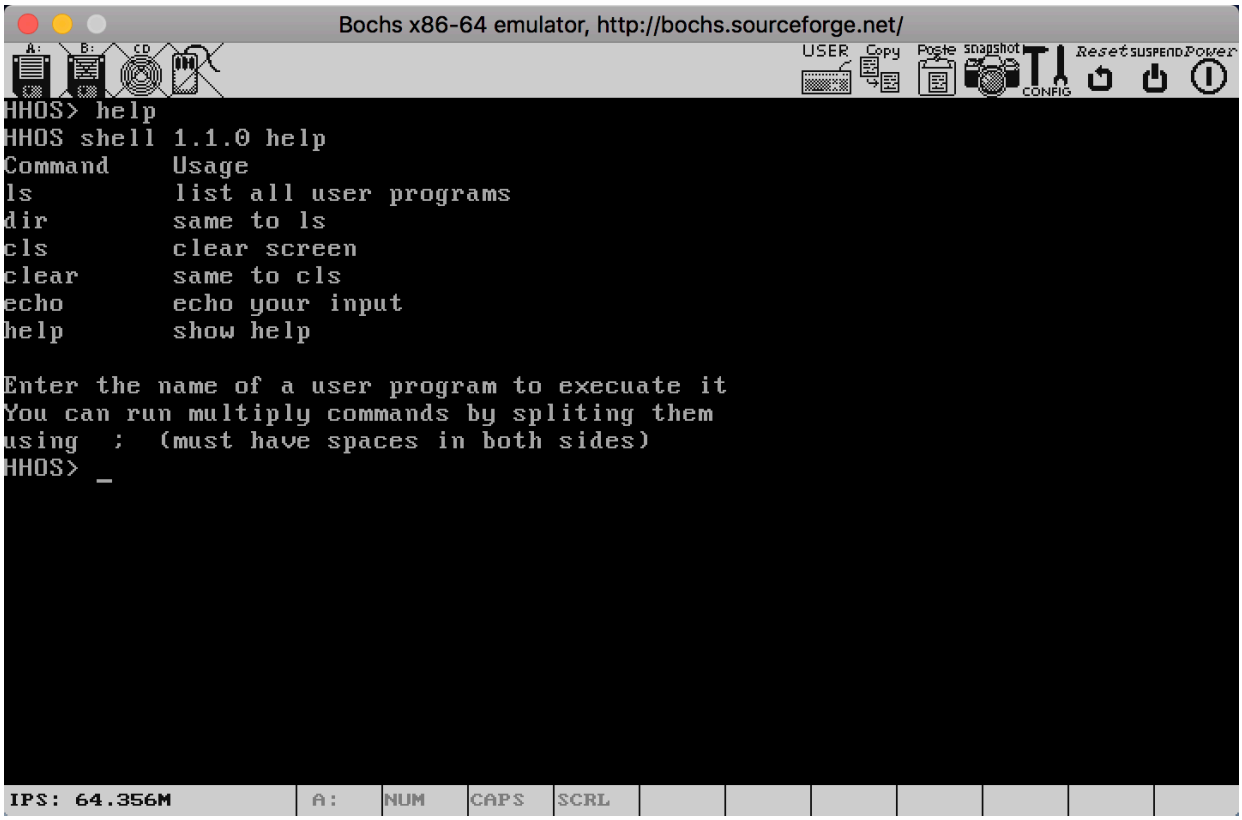


图五：调用计算器程序执行混合四则运算运算



图六：调用实验二的贪吃蛇程序

图七：显示帮助信息



五、实验总结

本次实验耗费了我半个月来几乎全部的可以自由安排的时间，令我感想颇多。

回首半个月前我的“监控程序”还只是一个只能根据长度限制为一个字符的输入，读取软盘（仅限前18个扇区）并放进内存的小程序。

半个月来，我通过广泛查询图书馆、网络上的资料，我自主探索出在Mac OS上交叉编译用于写操作系统的g++的方法、相关编译链接指令、nasm汇编与C++的互相调用规则，设计出操作系统的整体架构、分离出内核、移植封装实验二中的内核级函数，并实现更多内核函数，然后基于此实现tty模块、实现系统调用、实现并测试C语言库。实现加载用户程序模块、最后实现交互终端、写一个验证性的计算器程序。整个操作系统达到了34个文件，代码量达到了2500行。

Language	files	blank	comment	code
C++	10	67	157	1118
C/C++ Header	10	77	7	670
Assembly	12	70	15	627
make	2	66	13	125
SUM:	34	280	192	2540

这些步骤每一步都多多少少遇到一些问题。操作系统编程的一个特点是，问题出现时难以定位问题的源头在哪里，一个C++程序出现了问题，如果是大一的程序设计课程，就只需要调试程序逻辑。然而在写操作系统时，从基础的内核函数，到系统调用，到C函数库，每一个部分都是自己写的，如果每个层次不经过全面的测试，都不能肯定是不是那里其中有些隐藏的bug，最后导致了问题的出现。

我的计算器程序就出现过很奇怪的，可以进去，显示欢迎语句，然后就卡在那儿的情况。于是我就翻来覆去从上检查到下。结果最后发现哪儿都没没问题，就是程序写的比较长，读扇区读少了。

还有一个比较波折，令我印象最深刻的是加载用户程序时读软盘的问题。

有一天晚上在食堂吃饭，我心想写加载程序应该挺简单，之前实验二已经有相关代码，我只需要封装一下，十分钟就写完了吧？就拿出电脑开始写。当时是七点整，刚开始播新闻联播。这次实验里我把用户程序设计在从软盘第32号逻辑扇区开始放置，一开始我不知道软盘的结构，就把实验二里面调用bios中断时加载软盘的扇区号直接改为32，所以就什么都没有读到。

我百思不得其解，直到发现了bochs的警告信息。然而并不是很懂这个警告的意思，使用google搜索这个信息没得到解释。最后我只好找到了bochs源代码，在里面找到了这句报错，结合相关源代码的上下文才知道怎么回事。最终解决了这个问题时，我一听，怎么开始播新闻联播了（已经九点了😂）。

期待在接下来的实验中我能够让我的操作系统变得更强。

六、参考文献

1. 软盘结构及软盘数据的读取 https://blog.csdn.net/smallmuou/article/details/6796867
2. Memory Map (x86) https://wiki.osdev.org/Memory_Map_(x86)
3. OS Dev - C++ https://wiki.osdev.org/C%2B%2B
4. OS Dev - C Library https://wiki.osdev.org/C_Library
5. Linux内核完全剖析 https://book.douban.com/subject/3229243/
6. GCC Cross-Compiler https://wiki.osdev.org/GCC_Cross-Compiler