实验三：C与汇编开发独立批处理的内核

计算机科学与技术 18340146 宋渝杰

实验目的：

1. 加深理解操作系统内核概念
2. 了解操作系统开发方法
3. 掌握汇编语言与高级语言混合编程的方法
4. 掌握独立内核的设计与加载方法
5. 加强磁盘空间管理工作

实验要求：

1. 知道独立内核设计的需求
2. 掌握一种x86汇编语言与一种C高级语言混合编程的规定和要求
3. 设计一个程序，以汇编程序为主入口模块，调用一个C语言编写的函数处理汇编模块定义的数据，然后再由汇编模块完成屏幕输出数据，将程序生成COM格式程序，在DOS或虚拟环境运行。
4. 汇编语言与高级语言混合编程的方法，重写和扩展实验二的的监控程序，从引导程序分离独立，生成一个COM格式程序的独立内核。
5. 再设计新的引导程序，实现独立内核的加载引导，确保内核功能不比实验二的监控程序弱，展示原有功能或加强功能可以工作。
6. 编写实验报告，描述实验工作的过程和必要的细节，如截屏或录屏，以证实实验工作的真实性

实验内容：

1. 寻找或认识一套匹配的汇编与c编译器组合。利用c编译器，将一个样板C程序进行编译，获得符号列表文档，分析全局变量、局部变量、变量初始化、函数调用、参数传递情况，确定一种匹配的汇编语言工具，在实验报告中描述这些工作。
2. 写一个汇编程和c程序混合编程实例，展示你所用的这套组合环境的使用。汇编模块中定义一个字符串，调用C语言的函数，统计其中某个字符出现的次数，汇编模块显示统计结果。执行程序可以在DOS或虚拟机中运行。
3. 重写实验二程序，实验二的的监控程序从引导程序分离独立，生成一个COM格式程序的独立内核，在1.44MB软盘映像中，保存到特定的几个扇区。利用汇编程和c程序混合编程监控程序命令保留原有程序功能，如可以按操作选择，执行一个或几个用户程序、加载用户程序和返回监控程序；执行完一个用户程序后，可以执行下一个。
4. 利用汇编程和c程序混合编程的优势，多用c语言扩展监控程序命令处理能力。
5. 重写引导程序，加载COM格式程序的独立内核。
6. 拓展自己的软件项目管理目录，管理实验项目相关文档

实验过程：

1. 操作环境：
    - 操作系统：win10 + Linux
    - 虚拟机：VirtualBox
    - C 编译器：gcc
    - x86 编译器：nasm
    - 链接器：ld

    在本次实验中，我选择了 gcc + nasm 的组合，之所以不使用 tcc + tasm 是因为该组合版本太老了，即使能应付实验，总会被时代淘汰的

    样板 C 程序如下：（文件为 3-1.c）

```
int a=3,b=4,c;
void f(int,int);
void cmain(){
    f(a,b);
}
void f(int u,int v) {
    int d;
    c=u+v;
    d = 2;
}
```

其中定义了全局变量 a, b, c，局部变量 d，变量初始化 a=3，b=4，d=2，函数 f, cmain ， cmain 函数中调用 f 函数，传递参数 a 和 b

Linux 下使用编译行 `gcc -m16 -S -lntel 3-1.c -o 3-1.s` 编译得到 3-1.s 文件（该命令 win 也适用）

全局变量 b 关键代码及分析（ 本文的代码分析形式主要为注释 ）：

```
    .globl  b                   ; 全局变量
    .align 4                    ; 内存地址分配
    .type   b, @object          ; 类型：object
    .size   b, 4                ; 变量大小：4
b:
    .long   4                   ; 变量赋值：4
```

局部变量 d 和变量 d 初始化关键代码及分析：

```
    subl    $16, %esp           ; 声明：指针偏移
    ...
    movl    $2, -4(%ebp)        ; 赋值
```

函数 f 调用和参数传递关键代码及分析：

```
    subl    $8, %esp            ; 指针偏移
    movl    b, %edx
    movl    a, %eax
    subl    $8, %esp
    pushl   %edx                ; 传参
    pushl   %eax
    call    f                   ; call 调用
    addl    $16, %esp           ; 指针偏移返回
```

确定匹配的汇编语言工具：nasm

2. 汇编程序实例关键代码：（ 文件为 3-2asm.asm ）

   该模块为入口部分，定义和赋值全局变量 find，声明外部函数 main，之后直接调用 c 模块部分的 main 函数

```
    bits 16
    extern main                         ; main 在 c 程序中声明
    global find                         ; 声明全局变量 find ，给 c 程序调用
_start:
    call main                           ; 调用 c 程序 main
_end:
    jmp $

find:
    db "abbcccddddeeeeesyj",0x00     ; 字符串声明
    ; 注意：汇编显示信息在 c 程序中使用 asm 汇编内嵌编程实现
```

c 程序实例关键代码： （文件为 3-2c.c ）

该模块主要为函数实现部分，定义了 main 函数，用于统计字符串中相应字符的个数，并声明辅助函数 pchar 和 gchar，使用 asm 汇编内嵌技术，用于在键盘上接收一个字符，并输出在屏幕上

```
extern char find[];             // find[] 在汇编程序中声明

void pchar(char c) {
    ...
    asm volatile(               // asm 汇编内嵌编程，在屏幕上显示字符
        "push es\n"
        "mov es, ax\n"
        "mov es:[bx],cx\n"
        "pop es\n"
        :
        : "a"(0xB800), "b"((X*80+Y)*2), "c"((0x07<<8)+c)
        :);                     // a：显存起始地址 b：行列位置 c：颜色和字符
    ...
    asm volatile(
        "int 0x10\n"            // asm 汇编内嵌编程，设置光标位置
        :
        : "a"(0x0200), "b"(0), "d"((X<<8)+Y)); // a：功能号 b：页码 c：位置
    return;
    ...
}

char gchar() {
    char ch;
    asm volatile("int 0x16\n"   // asm 汇编内嵌编程，从键盘获取一个字符
                 : "=a"(ch)     // 获取的字符放入 ch 中
                 : "a"(0x1000));
    return ch;
}
    ...
void main() {
    ...
    char c = gchar();
    int num = 0;
    for (int i=0; i<18; ++i) {
        if (c == find[i]) num++;
    }
    ...
    pchar(c);
    c = (char)(num+48);
    ...
```

```
    pchar(c);
    return main();              // 实现重复统计
}
```

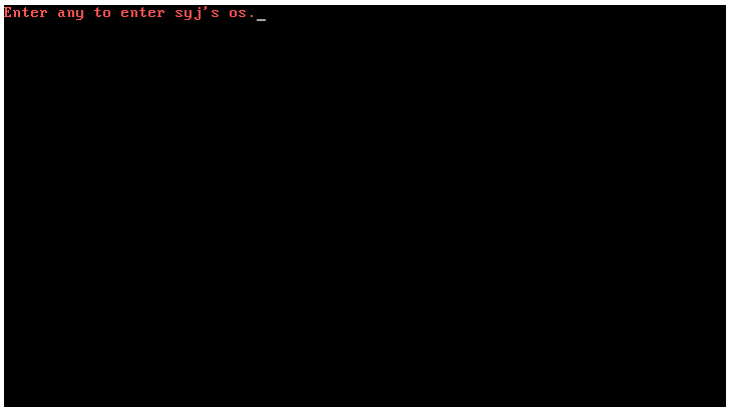
这里使用 asm 汇编内嵌编程而不直接在汇编程序实现显示效果，有许多优势所在：

- 本质不变：使用汇编代码显示信息，且显示效果相同
- 在 c 程序里面写汇编代码量比直接在汇编程序写要少，声明并输出多个提示信息字符串的代码要简洁太多
- 显示信息的代码可以重复调用，节省许多代码空间
- 可以在 c 程序中直接通过汇编模块显示信息，避免信息压栈传递带来的麻烦，也避免了编程时两边程序互相跳转查看对比带来的编程失误
- 可以成为一个实验创新点

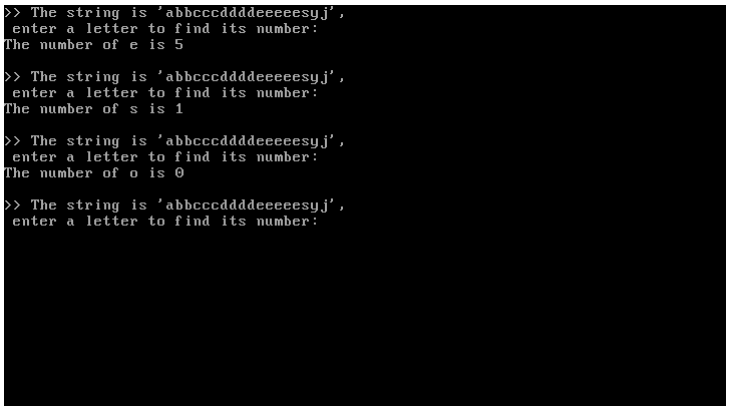
由于该代码长度超过 512 字节，故需要新建一个引导程序（文件为 3-2os.asm），代码和之后步骤 3，步骤 5 的引导程序一模一样，将会在下文介绍

win 下使用 gcc + nasm + ld 编译之后，放入 Linux 系统进行烧盘，最后得到 3-2.img 文件（编译命令行在下文叙述），测试效果如下：

刚进入操作系统：（带有闪烁效果）



进入用户程序后，分别按下 e, s, o 后，显示统计数据：（可多次统计）



3. 将实验二的监控程序分离成独立内核的具体步骤如下：
    1. 删去 `org 7c00h` ，使其离开引导扇区程序
    2. 仅保留 int 20h 中断部分、 清屏函数部分和 int 13h 加载扇区部分，其余部分全部用 c 实现
    3. 使用 `extern main` 标示 main 函数在 c 程序声明，同时汇编程序将 int 20h 中断加入中断向量表后，直接 `call main` 进入 c 模块
    4. 修改 int 20h 中断代码，使其返回 main 函数
    5. 使用 `global LoadnEx` 声明全局函数，供 c 程序调用，同时添加部分代码，使得其接收 c 函数压栈返回的值，并通过该值指定加载哪一块扇区

下面为关键代码：（其中包含 c 函数返回值压栈传递给汇编模块的技术）

```
    push ebp                    ; ebp 入栈
    mov ebp, esp                ; 用 ebp 来存取堆栈
    mov ecx, [ebp+8]            ; c 函数返回值赋值给 cx
    mov ch, 0                   ; 柱面号：起始编号为 0
    ...
    mov dh, 1                   ; 磁头号为 1，seek 从 18 开始
```

删去的功能，通过 c 模块来实现，具体步骤如下：

1. 添加 `extern void LoadnEx(int );` 标示 LoadnEx 函数在汇编程序声明
2. 编写 main 函数，关键代码如下：

   该代码先定义提示字符串并输出提示，之后从键盘读取一个字符，判断其合法后，根据字符数值选择相应的用户程序进行调用，最后返回本身，进行循环调用操作

```
void main() {
    ...
    static char msg1[] = "Welcome to syj's os! Enter a number (1-4) to
open a program: ";                      // 提示信息
    char *ptr = msg1;
    pstr(ptr);                  // 在屏幕打印提示信息
    ...
    char c = gchar();           // 从键盘读取一个字符
    while (c > '4' || c < '1') c = gchar();
    LoadnEx((int)(c-48));       // 加载汇编模块中的"加载扇区"函数
    return main();              // 循环
}
```

   其中的 I/O 操作均由 asm 汇编内嵌编程技术实现，关键代码如下：

```
/*
    其中 pchar 和 gchar 的关键代码在上文已提及
    读入/输出字符串操作均由循环读入/输出单个字符实现
*/
void pstr(char *s) {
    for (; *s; ++s)
        pchar(*s);              // 通过循环 pchar 来实现输出字符串
}

void gstr(char *s) {
    for (;; ++s) {
        pchar(*s = gchar());    // 循环 gchar 并通过 pchar 显示输入的字符
        if (*s == '\r' || *s == '\n')
            break;
    }
    *s = '\0';
}
```

分离出内核的引导扇区程序将会在步骤 5 介绍（文件为 3-3os.asm）

win下通过以下编译命令行生成 .o 文件并连接生成 .bin 文件：

```
    gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -fno-
PIE -masm=intel -c 3-3c.c -o 3-3c.o
    nasm -felf 3-3asm.asm -o 3-3asm.o
    ld -m elf_i386 -N --oformat binary -Ttext 0x7e00 3-3asm.o 3-3c.o -o 3-
3.bin
    nasm 3-3os.asm -o 3-3os.bin
```

对一些参数的解释如下：

- gcc：`-march=i386` 使用 i386 指令集；`-m16` 生成 16 位代码；`-mpreferred-stack-boundary=2` 栈指针按照 2*2=4 字节对其；`-ffreestanding` 使得输出程序能够独立运行；`-fno-PIE` 将程序编译成位置无关，并处理成 ELF 共享对象；`-masm=intel` 使用 nasm 格式的内联汇编语法
- nasm：`-felf` 把内核汇编模块编译成 ELF 文件
- ld：`elf_i386` 使用 i386 指令集处理 ELF 对象，`0x7e00` 为处理后 bin 内存位置

生成 3-3os.bin 和 3-3.bin 文件（即引导扇区程序和监控程序）后，在 Linux 下使用如下命令将这两个程序和实验二的四个用户程序烧进 img 文件里：

```
    /sbin/mkfs.msdos -C 3-3.img 1440
    dd if=3-3os.bin of=3-3.img conv=notrunc
    dd if=3-3.bin of=3-3.img seek=1 conv=notrunc
    dd if=1.com of=3-3.img seek=18 conv=notrunc
    dd if=2.com of=3-3.img seek=19 conv=notrunc
    dd if=3.com of=3-3.img seek=20 conv=notrunc
    dd if=4.com of=3-3.img seek=21 conv=notrunc
```

对应的一些技术细节如下：

- 1.com~4.com 为实验二的四个用户程序，代码只需要将 `org 100h` 改为 `org A100h`，使得其和监控程序读取扇区分配的内存地址相符
- 考虑到未来操作系统内核会扩大，在此直接分配 18 个扇区给引导扇区程序和内核（体现在监控程序读取扇区时 ch = 1，扇头号为 0 的所有扇区留给内核备用），因此 seek 从 18 开始计数

生成 3-3.img 后，可实现原来的 “从键盘输入一个字符，监控程序加载对应扇区的用户程序，运行一段时间后返回监控程序的功能”，部分实验效果截图如下：

通过引导扇区程序进入监控程序后显示提示消息：

键盘输入 2 ，执行右上角反弹程序，一段时间后停止并返回监控程序，重新显示提示消息：

4. 以步骤三得到的程序为基础，可以通过 c 模块定义更多指令，来扩展监控程序命令处理能力

   扩展后的指令定义和功能如下：

   - help：显示所有指令及功能
   - exe：执行一个程序（系统提示输入一个字符，根据字符调用相应用户程序）
   - exes：执行多个程序（系统提示输入四个字符，根据字符先后调用相应用户程序）
   - ls：列出所有用户程序信息
   - cls：清屏（通过汇编实现）
   - exit：结束监控程序（表现为之后对任何操作均无反应）

   修改 c 程序，具体步骤如下：

   1. 增加判断字符串是否完全相同函数：完全相同返回 0，否则返回 1

      ```
      int cstr(const char *s1, const char *s2) {
          while (*s1 && (*s1 == *s2))     // 逐位比较
              ++s1, ++s2;
          return (int)*s1 - (int)*s2;
      }
      ```

   2. 声明许多 char 字符串，用于显示新的提示信息，以及用于上述函数比较：

      ```
      static char msg1[] = "Welcome to syj's os! Enter a instruction (or
      Enter 'help' for help): \n";
      static char msg2[] = "Enter a number (1-4) to open an exe: \n";
      static char msg3[] = "Enter 4 number (1-4) to open 4 exes: \n";
      static char err[] = "Error: Instruction does not exist!\n";
      const char help[] = "help", exe[] = "exe", exes[] = "exes", exit[] =
      "exit", ls[] = "ls", clear[] = "cls";
      static char helpstr[] =
          "You can input these instructions.\n"
          "\n"
          "help         -- Print instructions\n"
          "exe          -- Open an exe\n"
          "exes         -- Open many exes\n"
          "ls           -- Show four exes' information\n"
          "cls          -- Clear the screen\n"
          "exit         -- Exit OS\n";
      static char exestr[] =
          "exe1         -- LeftUp     512 bytes\n"
          "exe2         -- RightUp    512 bytes\n"
          "exe3         -- LeftDown   512 bytes\n"
          "exe4         -- RightDown  512 bytes\n";
      ```

3. 系统输入改为 `pstr(ptr);`，直接输入字符串当作指令进行判断
4. 定义辅助变量，用于指令功能的实现，同时声明外部定义函数 cls，由汇编模块之前步骤的清屏函数来实现清屏功能（注意将汇编模块的 cls 函数加上 global 声明）：

```
int X = 0, Y = 0, num = 4;
int queue[] = {0,0,0,0};            // 程序顺序执行队列
extern void cls();                  // 同时在汇编模块用 global 声明 cls 函数
```

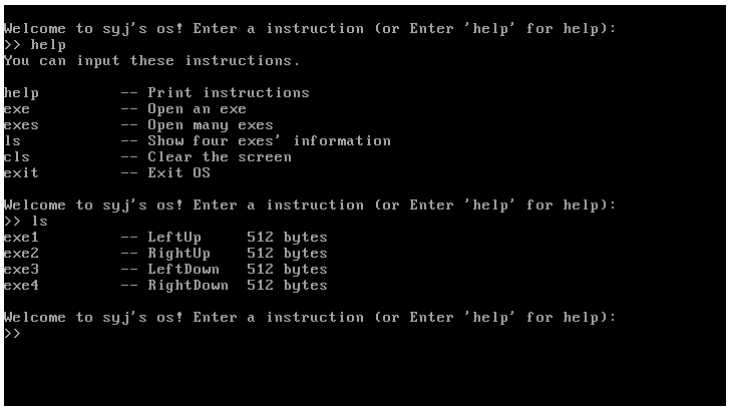
5. 对输入的字符串指令进行判断：

```
if (cstr(str, help) == 0) {         // 如果指令是 help
    ptr = helpstr;
    pstr(ptr);                      // 输出 help 提示字符串
}
else if (cstr(str, clear) == 0) {   // 如果指令是 cls
    cls();                          // 调用汇编清屏函数
    X = 0;
    Y = 0;                          // 重新定位提示信息位置
}
else if (cstr(str, exit) == 0) {    // 如果指令是 exit
    return;
}
else if (cstr(str, ls) == 0) {      // 如果指令是 ls
    ptr = exestr;
    pstr(ptr);                      // 输出 ls 提示字符串
}
else if (cstr(str, exe) == 0) {     // 如果指令是 exe
    ptr = msg2;
    pstr(ptr);                      // 输出 msg2 提示字符串
    pchar('\n');
    pchar('>');
    pchar('>');
    pchar(' ');
    char c = gchar();               // 输入一个字符
    while (c > '4' || c < '1') c = gchar();
    LoadnEx((int)(c-48));           // 调用汇编读取扇区并加载用户程序
}
else if (cstr(str, exes) == 0) {    // 如果指令是 exes
    ptr = msg3;
    pstr(ptr);                      // 输出 msg3 提示字符串
    pchar('\n');
    pchar('>');
    pchar('>');
    pchar(' ');
    for (int i=0; i<4; i++) {       // 循环读入
        char c = gchar();           // 输入一个字符
        while (c > '4' || c < '1') c = gchar();
        pchar(c);                   // 输出输入的字符
        queue[i] = (int)(c-48);     // 装入队列
    }
    num = -1;                       // 初始化队列头
}
else {                              // 如果不存在该指令
    ptr = err;
    pstr(ptr);                      // 输出 err 提示字符串
```

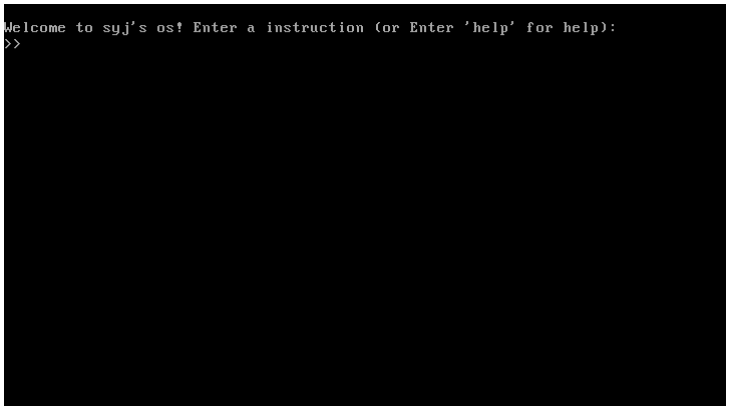
```
}
return main();                          // 循环
```

通过上文的编译命令行生成 3-4.img 文件，测试运行结果如下：

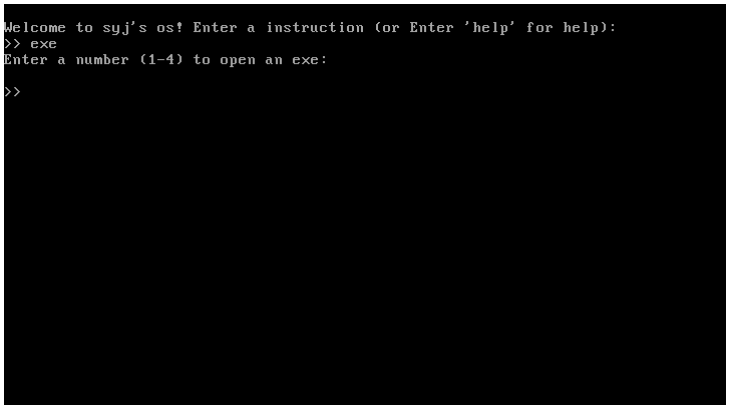
测试 help 和 ls 指令：



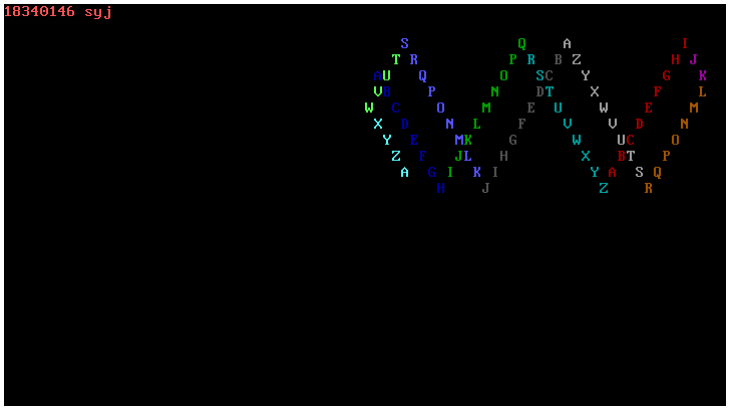
在上一步基础上测试 cls 指令：（输入 cls，回车后的效果图）



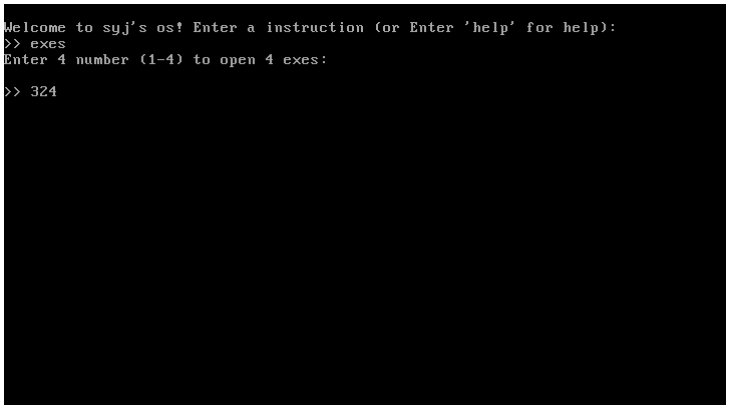
测试 exe 指令：



键盘输入 2 之后，执行用户程序 2：



测试 exes 指令，并按顺序输入 3241：（图片为输入到 4 时的截图）



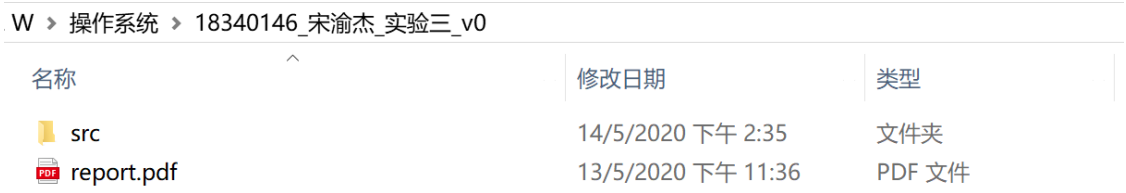
（exes 测试指令效果截图不好显示，请移步 3-4.img 自行测试）

5. 新的引导扇区关键代码和分析如下：

   该程序主要实现 int 13h 加载扇区并跳转即可，我在这个基础上加入了字符串信息提示和键盘上按键才跳转的功能，可以使得该程序更加美观

```
    bits 16                             ; 16 位编译
    monitorr equ 7e00H                  ; 监控程序地址
    org 7c00H                           ; 引导扇区程序
start:
    ...
    int 10h                             ; 显示提示字符串
    ...
    int 16H                             ; 按任意键跳转
load:
    ...
    mov al, 17                          ; 预留 17 个扇区供以后内核扩展需要
    int 13H                             ; 读取扇区
    call monitorr                       ; 这里采用 call，能实现正确跳转
    ...
msg:                                    ; 提示信息
    db "Enter any to enter syj's os."
```

6. 自己的软件项目管理目录如下图：



实验心得：

本次实验采取了较新的 gcc + nasm + ld 的方式，没有走老师的 tcc + tasm + tlink 的老路，其实是存在不少学习上的问题的：

- gcc + nasm + ld 网上和现实中参考资料极少，老师方面基本没有参考程序帮助，ppt 也没有许多知识点的内容，比如说在步骤二，c 模块函数返回值压栈传给汇编模块，老师和 ppt 根本没有这方面的代码参考，我只能在网上不断找有关代码参考，发现有更好的方式能通过汇编显示消息，就采取了该方法，马上跳到下一步骤研究，时间方面非常的紧迫
- gcc 参数问题：上文中讲到 gcc 编译命令行参数多而杂，我也是通过班群、网络等等渠道收集到一点一点的信息，进行拼接和测试，直到编译和链接均没有 error 报错，才能开始下一步骤研究

而对于其它技术层面的心得，我也在此一并讲述：

- 代码纠错：这次程序没有老师的代码参考，故没有老师代码纠错部分
- 实验中出现的出错问题及解决方式：一个问题出现在汇编的 global 声明：由于没有参考代码，我尝试了许多种 global 的声明格式，其中只有一种是能成功通过编译的：

```
; right
    global find
find:
    db "abbcccddddeeeeesyj",0x00

; err 1
; ld: error: undefined reference to `find'
    global find:
    db "abbcccddddeeeeesyj",0x00

; err 2
; nasm: error: identifier expected after GLOBAL
    global find db "abbcccddddeeeeesyj",0x00
```

还有一个问题是：在步骤二中，本来想通过 c 程序 asm 汇编内嵌技术输出汇编程序定义的字符串（即 find[] ）进行信息的提示，这个问题并没有想到方式解决，时间关系就采取了提示字符中直接加入输出统计的字符串信息（即直接加入 'abbcccddddeeeeesyj' 进行输出）

```
extern char find[];              // 外部定义需要统计的字符串
char msg[] = "The string is 'abbcccddddeeeeesyj',\n enter a letter to find
its number: \n";                 // 信息中直接输出 'abbcccddddeeeeesyj'
for (int i=0; i<18; ++i) {
    if (c == find[i]) num++;     // 统计还是用 find 进行统计
}
```

- 技术替换：对于这次实验的步骤二，我采取了 “ c 程序中的 asm 汇编内嵌编程” 代替 “ c 程序函数返回值传递给汇编模块输出” 来实现统计信息的显示功能，本质上也是通过汇编来实现信息的显示，显示的效果也相同，但是，当我们接通了 c 之后，熟悉了 asm 汇编内嵌编程之后，换一种方式实现信息显示是非常简洁且高效的
- 在步骤三：内核分离中，我把可以用 c 实现的功能基本都交给 c 模块，只把不得不用汇编实现的中断、读取扇区，和一个汇编代码也比较简单的清屏留在了汇编模块，这样处理代码能变得极为简洁且易于理解和处理
- 在步骤四中，接入了 c 以后，实现输入字符串和字符串处理变得简单了许多，因此只需要对字符串做对比匹配操作，即可实现许多命令的自创

而对于非技术层面的问题，我也有不少感触：

- 本次实验难度过大，实验量严重超出一周时间范围，基本没有任何现成参考，路线又有分歧，以至于班群几百条问题信息，朋友圈不少同学都存在负面情绪
- 这次实验花费至少完整的 3 天时间去搜索资料、研究、编写代码、调试、编写报告是完全不奇怪的，所以请关爱一下那些光专业课就有 6（甚至最高到 8 ）门的同学们，请尊重同学们的劳动成果