实验三实验报告

一、 【个人信息】

院系：数据科学与计算机学院
专业：计算机科学与技术（超算方向）
年级：2016
班级：教务 2 班
姓名：劳马东
学号：16337113
邮箱：laomd@mail2.sysu.edu.cn

二、 【实验题目】

开发独立内核的操作系统：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，存放在其他扇区中，为以后扩展内核提供发展空间。
操作系统内核：

- 可加载多个用户程序
- 汇编模块
- C 模块
  - 在磁盘上建立一个表，记录用户程序的存储安排；
  - 可以在控制台查到用户程序的信息，如程序名、字节数、在磁盘映像文件中的位置等；
  - 设计一种命令，并能在控制台发出命令，执行用户程序；

三、 【实验目的】

1. 学习 C 与汇编混合编程，掌握 GCC+NASM 交叉编译技术；
2. 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。

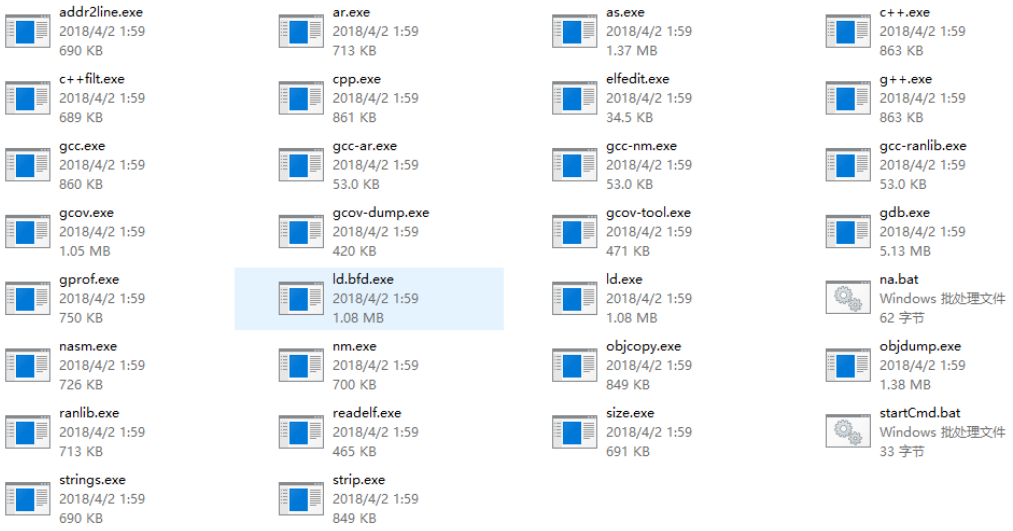
四、 【实验要求】

1. 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。
2. 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。
3. 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。

五、 【实验方案】

(一) 实验工具：GCC+NASM

交叉编译工具链：



1. GCC 编译生成链接文件
   gcc -Og -c <cfile>.c -o <cfile>.o
2. NASM 汇编命令
   1) 生成 com 文件
      nasm -f bin <afile>.asm -o <afile>.com
   2) 生成链接文件
      nasm -f elf32 <afile>.asm -o <afile>.o
3. 链接命令
   1) 生成引导扇区程序：
      ld -N <afile>.o <cfile>.o -Ttext 0x7c00 --oformat
          binary -o <output>.bin
   2) 生成 COM 程序：
      ld -N <afile>.o <afile>.o -Ttext 0x100 --oformat
          binary -o <output>.com

(二) 实验原理

1. 编译参数

GCC 编译 C 内核代码例子：

```
$ gcc -c -ffreestanding -m32 -march=i386 -mpreferred-stack-boundary=2 \
    -o kernel.o kernel.c
```

使用 GCC 编译时需要传入下表中的参数。

参数	说明
-c	只编译不链接
-ffreestanding	使输出程序能独立运行
-m32（非x86_64 的 GCC 不需要）	生成 32 位代码（16 位也需要此参数，见4.1节）
-march=i386	使用 i386 指令集
-mpreferred-stack-boundary=2	栈指针按 $2^2 = 4$ 字节对齐
-fno-exceptions（非 C++ 不需要）	不使用异常处理
-fno-rtti（非 C++ 不需要）	不使用 RTTI
-o XXX.o	输出文件名
其他	输入文件名

2. 链接参数

```
$ ld -melf_i386 -ffreestanding -nostdlib -N -Ttext=0x0500 \
    --oformat=binary -o kernel.bin kernel.o utils.o
```

参数解释如下表。

参数	说明
-melf_i386	指定输入的目标文件格式
-ffreestanding	使输出程序能独立运行
-nostdlib	不使用标准库
-N	关闭页对齐，但分页内核不能关闭
-Ttext=0x0500	指定加载位置，根据需要设置地址
--oformat=binary	指定输出格式为 Binary（Flat），根据需要指定
-o XXX.o	输出文件名
其他	输入文件名

3. GCC 与 NASM 交叉编译

1） 汇编模块中调用 C 模块中的函数

- 调用前要用 extern 声明 C 模块的函数
- 根据 C 中函数原型，用栈传递参数，顺序后参先进栈
- 调用 C 函数后，要将栈中参数弹出
- 进栈出栈以 4 字节为单位
- 返回地址为 4 个字节，因此除了传递参数，在 call 之前必须把 cs 寄存器的值入栈（cs+ip 总共 4 个字节）

2） C 模块中调用汇编函数和引用变量

- 汇编模块的过程从栈中取得参数，不必出栈，直接引用栈中的值，顺序与 C 进栈对应
- 由于 NASM 的 ret 指令只取栈顶两个字节（ip），会导致从汇编函数返回时在栈顶残留两个字节，因此要把 ret 指令替换为 retf

（相当于 pop ip 和 pop cs）或者加入 32 位操作数前缀：o32 ret

- 如果 C 中想引用汇编模块中的变量和标识符，汇编模块中要用 global 声明这些符号，C 中定义的函数默认是 global 的

六、 【实验过程】

1. 命令行选项与新汇编语句理解

1) -Ttext addr

以 elf32 模式生成汇编.o 文件时，org 指令被禁用，如下图：

```
1  org 7c00h
2  extern myUpper
3  extern myMessage
4  global _start
5  _start:
```

```
D:\教材\大二下\操作系统\实验\project3\3gcc>nasm -f elf32 afile.asm -o afile.o
afile.asm:1: error: parser: instruction expected
```

链接时通过-Ttext addr 的方法指定入口地址，即-Ttext 是链接时将初始地址重定向为 0x7c00(若不注明此,则程序的起始地址为 0)。

2) global _start

汇编中的 global _start 是必要的，因为使用-Ttext addr 的方式指定入口地址需要有一个可访问的_start 标签作为入口，因此代码从 addr:_start，否则将默认以 addr:0 作为入口。

3) BITS

汇编开头使用了 BITS 16 指令，我们要告诉汇编器我们的程序要被当作 16 位的程序还是 32 位的程序来运行，好让它汇编出正确的机器码，BITS 伪指令的作用就在于此。看一个简单的程序:

```
1  BITS 16
2  mov ax, 1
3  mov eax, 1
4  jmp $
```

```
1  BITS 32
2  mov ax, 1
3  mov eax, 1
4  jmp $
```

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	B8	01	00	66	B8	01	00	00	00	EB	FE					

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	66	B8	01	00	B8	01	00	00	00	EB	FE					

当 NASM 在 BITS 16 状态下时，使用 32 位数据的指令要加一个字节的前缀 0x66，要使用 32 位的地址，则加上 0x67 前缀。在 BITS 32 状态下，相反的情况成立，32 位指令不需要前缀，而使用 16 位数据的指令需要 0x66 前缀，使用 16 位地址的指令需要 0x67 前缀。

4) __asm__(".code16gcc\n")

C 语言程序的开头使用了__asm__(".code16gcc\n") 嵌入汇编指令，以指示 as 生成 16 位代码。有什么区别？编写一个简单程序来测试：

```
1   __asm__(".code16gcc\n");
2
3   int a = 0;
4   void f()
5   {
6       a++;
7   }
```

```
1   int a = 0;
2   void f()
3   {
4       a++;
5   }
```

分别用 objdump 反汇编生成的.o 文件，如下：

```
D:\教材\大二下\操作系统\实验\project3\3gcc>gcc -march=i386 -m32 -mpreferred-stack-boundary=2
-ffreestanding -c cfile.c -o cfile.o

D:\教材\大二下\操作系统\实验\project3\3gcc>objdump -D cfile.o
```

加嵌入汇编和不加的结果如下：

```
cfile.o:     file format elf32-i386


Disassembly of section .text:

00000000 <f>:
   0:   66 55                   push   %bp
   2:   66 89 e5                mov    %sp,%bp
   5:   66 a1 00 00 66 40       mov    0x40660000,%ax
   b:   66 a3 00 00 90 66       mov    %ax,0x66900000
  11:   5d                      pop    %ebp
  12:   66 c3                   retw
```

```
cfile.o:     file format elf32-i386


Disassembly of section .text:

00000000 <f>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   a1 00 00 00 00          mov    0x0,%eax
   8:   40                      inc    %eax
   9:   a3 00 00 00 00          mov    %eax,0x0
   e:   90                      nop
   f:   5d                      pop    %ebp
  10:   c3                      ret
```

可以看到，使用__asm__(".code16gcc\n") 嵌入汇编指令，编译器生成的代码使用的是 bp、sp、ax 等 16 位寄存器，返回指令时 retw(return word)16 位指令；而不使用指令，代码使用 ebp、esp、eax32 位寄存器，返回指令时 ret（32 位）。因此，要想 GCC 生成的.o 和 NASM 生成的.o 能正确地相互调用，需要加上这一汇编指令，并在 NASM 中加 BITS 16 指令。

另外，调用 gcc 时除了指定-c 选项指示它只编译不连接外，还要指定-m32 选项，这样才会生成 32 位的汇编代码，而只有在 32 位的汇编代码中使用.code16gcc 指令，才能编译成 16 位的机器码。如果没有指定-m32 选项，则生成的是 64 位汇编代码，然后汇编时会出错。使用-m32 选项后，生成的目标文件是 ELF32 格式。ELF32 格式的目标文件只能和 ELF32 格式的目标文件连接，这也是为什么前面的 nasm 和 ld 需要指定 elf32 和-m elf_i386 选项。

2. C 与汇编之间相互调用的探索

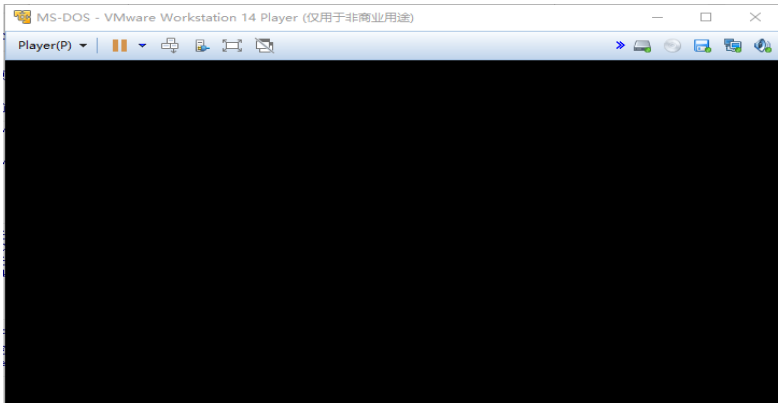
1) 返回地址

A. 汇编调用 C

有几行代码让人感到疑惑：

```
1   BITS 16
2   extern myUpper
3   extern myMessage
4   global _start
5   _start:
6       mov ax,cs
7       mov ds,ax
8       mov es,ax
9       mov ss,ax
10  Load:
11      mov bx,7e00h
12      mov ah,2                ; 功能号
13      mov al,10                ;扇区数
14      mov dl,0                ;驱动器号 ; 软盘为0, 硬盘和U盘为80H
15      mov dh,0                ;磁头号 ; 起始编号为0
16      mov ch,0                ;柱面号 ; 起始编号为0
17      mov cl,2                ;起始扇区号 ; 起始编号为1
18      int 13H                 ;调用读磁盘BIOS的13h功能
19  _call:
20      push cs
21      call myUpper
22      mov bp,myMessage
23      mov ax,ds
24      mov es,ax
25      mov cx,10
26      mov ax,1301h
27      mov bx,0007h
28      mov dh,10
29      mov dl,10
30      int 10h
31  _end:
32      jmp $
```

为什么在调用 C 函数之前要把 cs 入栈？调用完之后为什么不出栈？注释掉 push cs，再次编译，结果惊人：



myUpper 没有正确返回！这是因为返回地址是 4 个字节，C 函数返回时，从栈顶取 4 个字节，但是栈顶只有 ip，因此 myUpper 返回时，取出 ip 和 ip 之下的两个字节（不确定的数）作为其返回

地址，于是就不知道跳到了哪里。所以 push cs 是必要的，这样才能正确返回到 cs:ip 的地方。

B. C 调用汇编

测试代码如下：

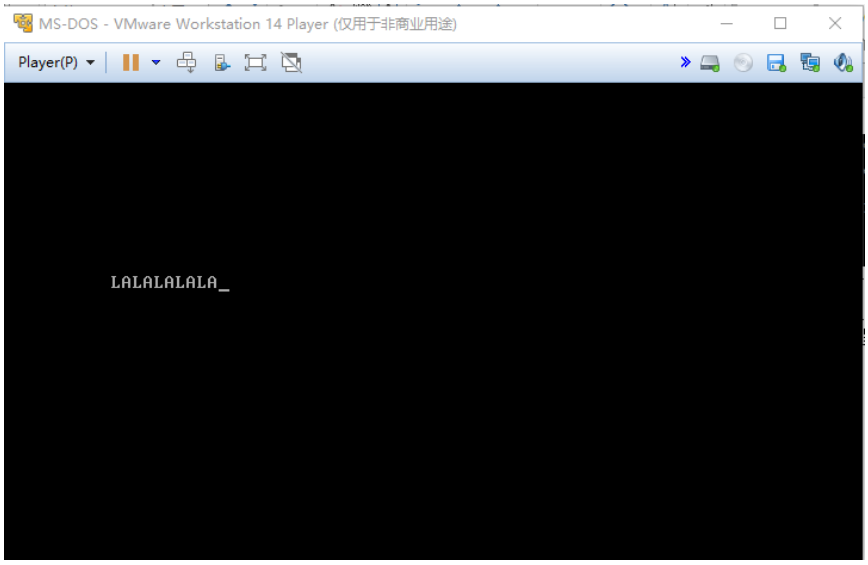
afile.asm

```
BITS 16
extern main
global _start
_start:
    mov ax,cs
    mov ds,ax
    mov es,ax
    mov ss,ax
Load:
    mov bx,7e00h
    mov ah,2                    ; 功能号
    mov al,10                    ;扇区数
    mov dl,0                    ;驱动器号 ; 软盘为0，硬盘和U盘为80H
    mov dh,0                    ;磁头号 ; 起始编号为0
    mov ch,0                    ;柱面号 ; 起始编号为0
    mov cl,2                    ;起始扇区号 ; 起始编号为1
    int 13H                     ;调用读磁盘BIOS的13h功能
    push cs
    call main
    jmp $

f:
    mov bp,myMessage
    mov ax,ds
    mov es,ax
    mov cx,10
    mov ax,1301h
    mov bx,0007h
    mov dh,10
    mov dl,10
    int 10h
    ret

myMessage db "LALALALALA"
```

cfile.c

```
GLOBAL f
f:
    mov bp,myMessage
    mov ax,ds
    mov es,ax
    mov cx,10
    mov ax,1301h
    mov bx,0007h
    mov dh,10
    mov dl,10
    int 10h
    ret
```

打开虚拟机结果正常：



可是真的没有问题吗？不妨多调用几次 f 函数。

afile.asm 更改如下：

```
GLOBAL f
f:
    ; mov bp,myMessage
    ; mov ax,ds
    ; mov es,ax
    ; mov cx,10
    ; mov ax,1301h
    ; mov bx,0007h
    ; mov dh,10
    ; mov dl,10
    ; int 10h
    ret

```

cfile.c 更改如下：

```
__asm__(".code16gcc\n");
extern void f();
void main()
{
    for (int i = 0; i < 10000; ++i)
    {
        f();
    }
}
```

编译打开虚拟机，runtime error:

原因是 main 函数调用 f 的时候，放入栈顶的返回地址是 4 个字节的，而 NASM 的 ret 语句只是在栈顶取出两个字节（ip）,因此每次循环，栈顶都残留两个字节的数据，如此下去，栈越来越大，很快就溢出了。怎么办呢？一个做法是 ret 之前把 ip 之下的两个字节 pop 出来，如下：

```
22  GLOBAL f
23  f:
24      pop ax  ;ip
25      pop bx  ;cs
26      push ax
27      ret
28
```

如此再次运行就没有问题了。可是这样做的缺点是需要改动 ax 和 bx，我们并不想在一个函数中更改不必要的寄存器。
另一个做法是用 **retf** 语句。retf 的效果等价于 ret 和一条 pop cs 语句，这正是我们需要的。当然，也可以在 ret 前加上 32 位前缀 o32。

```
22  GLOBAL f
23  f:
24      retf
25
```

2) 参数传递
编写一个简单的 C 程序，观察 GCC 编译产生的 AT&T 汇编代码（局部），代码如下：

```c
__asm__(".code16gcc\n");
int add(int,int);
void foo(char*, int);

int a = 3, b = 4;
char str[1];

void cmain(){
    int c = add(a, b);
    foo(str, c);
}

int add(int a,int b) {
    return a + b;
}

void foo(char* s, int ascaii)
{
    *s = ascaii;
}
```

编译命令：

```
D:\教材\大二下\操作系统\实验\project3\3gcc>gcc -masm=intel -m16 -Og -S test.c -o test.asm

D:\教材\大二下\操作系统\实验\project3\3gcc>
```

产生的汇编文件代码如下：

```
add:
.LFB1:
    .cfi_startproc
    mov eax, DWORD PTR [esp+8]
    add eax, DWORD PTR [esp+4]
    ret
    .cfi_endproc
```

```
foo:
.LFB2:
    .cfi_startproc
    mov eax, DWORD PTR [esp+4]
    mov edx, DWORD PTR [esp+8]
    mov BYTE PTR [eax], dl
    ret
    .cfi_endproc
```

```
30  cmain:
31  .LFB0:
32      .cfi_startproc
33      push    DWORD PTR b
34      .cfi_def_cfa_offset 8
35      push    DWORD PTR a
36      .cfi_def_cfa_offset 12
37      call    add
38      add esp, 8
39      .cfi_def_cfa_offset 4
40      push    eax
41      .cfi_def_cfa_offset 8
42      push    OFFSET FLAT:str
43      .cfi_def_cfa_offset 12
44      call    foo
45      add esp, 8
46      .cfi_def_cfa_offset 4
47      ret
48      .cfi_endproc
```

观察其中一些高亮的代码，可以发现一个规律：

A. C 语言程序中，GCC 编译后，变量名和 C 中一样，函数名也如此；

B. 参数传递时，字符串首地址和整数都是 4 个字节（DWORD）压栈，按后面参数先进栈，因此在汇编中取参数时，sp 或者 esp（NASM 用 bp 或 ebp）应该每次加 4 而不是 2，取到参数与 C 中传递的参数的意义一致。调用之后修正 esp 指针，消除栈中参数，因此在汇编函数中不用自己手动把参数出栈。

用图来解释：

```
90  ; ================================================================
91  ; void _delay(int h, int l);
92  ; ================================================================
93  GLOBAL _delay
94  _delay:
95      push ax
96      push cx
97      push dx
98      push bp
99          mov bp,sp
100         mov cx,[bp+4+2*4]
101         mov dx,[bp+4*2+2*4]
102         mov ah,86h
103         int 15h
104     pop bp
105     pop dx
106     pop cx
107     pop ax
108     retf
```

这是一段延时代码，延迟 cx:dx 微秒。2*4 是因为把 ax、cx、dx、bp 入栈，每个 2 个字节。取 cx 时+4 是因为 h 传进来的时候是 4 个字节，加了 4 才能让 bp 指向 h 的开始地址；取 dx 时加 4*2 是因为 l 在 h 之下 4 个字节的地方开始。

同理，汇编给 C 函数传参时，需要传 4 个字节，如要传递 ax，则应先 push 0 再 push ax，0 是参数的高 16 位，参数入栈的顺序与 C 函数声明中的顺序相反。

3. 从监控程序中分离操作系统内核

在实验二中，监控程序包含了引导程序和操作系统。为了把操作系统内核分离出来，只需要把操作系统作为一个用户程序，在引导程序中加载执行操作系统，并把控制权交给它。

实验二监控程序部分代码如下：

```
org  7c00h
OffSetOfUserPrg1 equ 8100h
start:
    mov ax, cs
    mov es, ax
    mov ds, ax
LoadnEx:
    call input
    cmp al,1
    jz clearWindow
    call callProgram
    jmp loopEnd
clearWindow:
    call clear
loopEnd:
    loop LoadnEx

callProgram:
    push ax
    mov ax,cs
    mov es,ax
    mov ds,ax
    mov bx, OffSetOfUserPrg1
    mov ah,2
    mov al,1
    mov dl,0
    mov dh,0
    pop cx
    mov ch,0
    int 13
    call bx
    ret
```

其中，起到引导程序作用的代码只是 org 7c00h 语句，它负责把代码加载到内存 7c00h 的地方，使得开机的时候系统跳到 7c00h 地址的时候可以正确执行以上代码。其余代码都是操作系统的部分，因此把以上代码分成两部分。

引导程序（局部）：

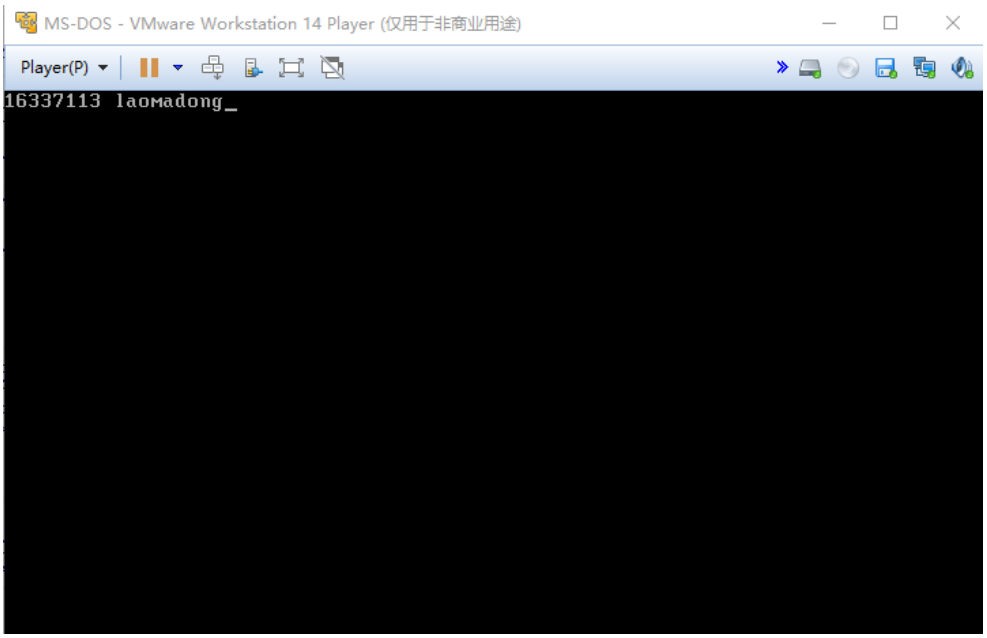
```
org 7c00h
_start:
    mov ax,cs
    mov ds,ax
    mov es,ax
    mov ss,ax
    push 7e00h
    push 10
    push 2
    call Load

_end:
    mov ah,4ch
    int 21h

Load:...
times 510-($-$$) db 0
dw 0xAA55
```

操作系统（局部）：

```
org  7e00h
OffSetOfUserPrg1 equ 8100h
start:
    mov ax, cs
    mov es, ax
    mov ds, ax
    call printString

printString:
    mov bp, Message
    mov cx, MessageLength
    mov ax, 1301h
    mov bx, 0007h
    mov dx, 0
    int 10h
    ret

Message:
      db '16337113 laomadong'
MessageLength  equ ($-Message)
x db 1
times 512-($-$$) db 0
```

操作系统放在从第二个扇区开始的 10 个扇区中，加载到内存的 7e00h 处。

引导程序的作用就是，从磁盘中读操作系统并加载到 7e00h，然后 call 到 7e00h 的地方，就把控制权交给了操作系统。操作系统的 org 是 7e00h，要做的事情就是打印个人信息。不妨测试一下！



测试结果符合预期，操作系统已经被分离出来了。之后扩展操作系统代码，和 C 交叉编译，让它加载并执行用户程序，就完成了一个操作系统内核原型。最终做出的操作系统代码详见源代码文件。

4. 编写启动代码

在 C 和汇编混合编程的时候，发现有一部分代码是经常要写的：

```
1   BITS 16
2   EXTERN main
3   GLOBAL _start
4   _start:
5       mov ax,cs
6       mov ds,ax
7       mov es,ax
8       mov ss,ax
9       call main
10      ret
```

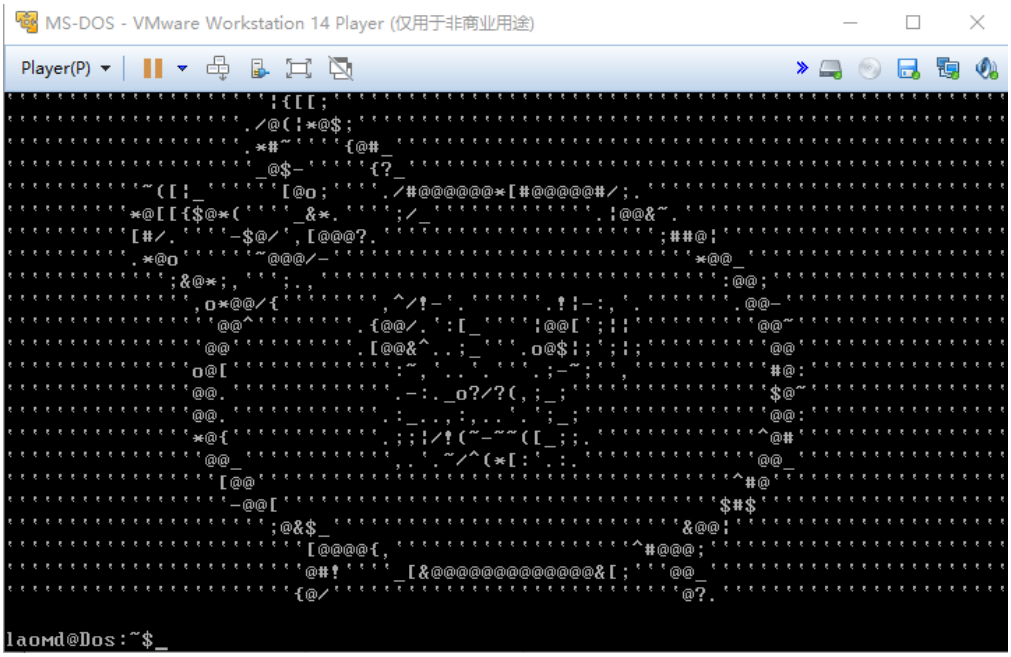
代码的作用是初始化一些段寄存器，然后调用 C 中的 main 函数，并在 main 函数返回时回到上一层调用者。通过查阅一些 GCC 文档，发现 C 程序在 main()执行之前还有启动代码，通常叫作 c0、crt0 之类，.exe 真正的入口点在这里面，它执行一系列的初始化工作再调用 main()。于是就把上面这段代码当做启动代码，编译成一个.o 文件，所有需要交叉编译的程序只需要链接这个.o 就可以。如下：

```
1  @echo off
2  set asmfile=os_a
3  set cfile=terminal
4  set LOADPOS=0x7e00
5  set LIBPATH=../lib
6  nasm -f elf32 %asmfile%.asm -o %asmfile%.o
7  gcc -fno-builtin -Werror -Og -c %cfile%.c -o %cfile%.o
8  ld -N %LIBPATH%/crt0.o %asmfile%.o %cfile%.o %LIBPATH%/klib.o -Ttext %LOADPOS% --oformat binary -o %asmfile%.com
9  del %asmfile%.o %cfile%.o
```

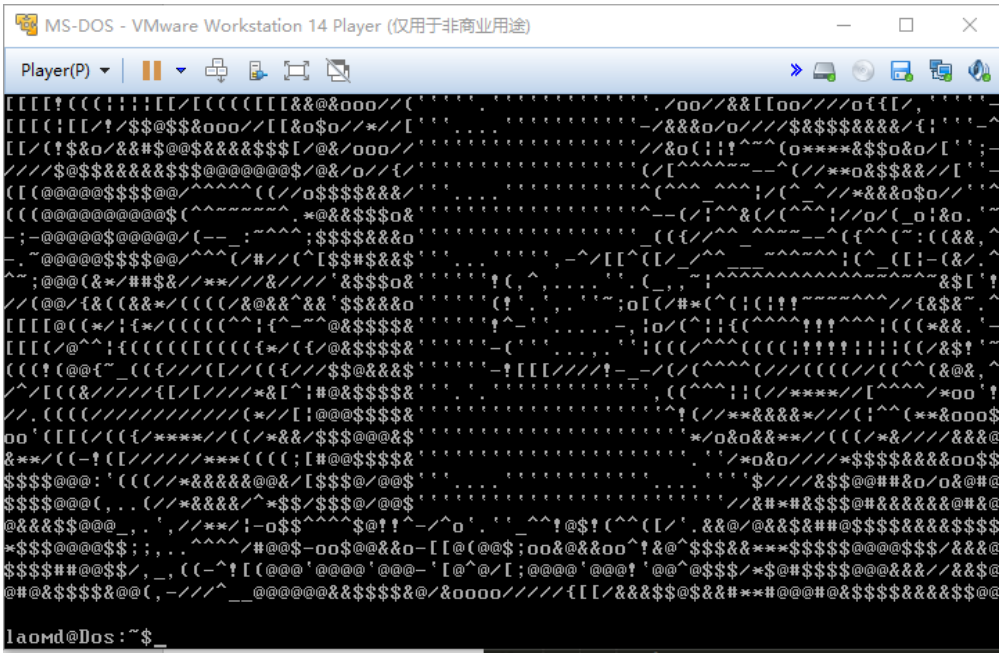
5. 编写用户程序

用户程序也是采用 GCC+NASM 混合编程的方式，因为在 C 中写代码方便很多。一共设计了 2 个用户程序，分别是兔子和 sorry 动图的打印。效果如下：

兔子动图：



Sorry 动图：



方法是把动图的每一帧转换成字符串，然后建立一个很大的字符串数组，每隔一段时间播放一帧对应的字符串。

```
294  void main()
295  {
296      int n = sizeof(frames) / sizeof(frames[0]);
297      for (int i = 0; i < n; ++i)
298      {
299          cls();
300          puts(frames[i]);
301          sleep(200);
302      }
303      putch('\n');
304  }
```

其中 frames 是字符串数组，部分内容截图如下：

```
6   char* frames[] = {
7   "[[[[///(!!!!|((![!!!![///[o$$&&&&//``.,.`````-/&/o*$$@@@$**o&&&&///{^.,.`````````^\
8   [[[[{!!!|||||!((!||||([[[&@@oo*o/&``.`````````-oo$&&&&$$$$&&&&$$$*&&o/o!`````````-\
9   //[[{!!!|||||!!!!(((((((((*@@oo/o//``.`.`````_{oo$&&**&o&&///*o//o/////[.:,,.`````-\
10  [[[[/||(!!!![[[[!!(![[[[/[/$$&/o&/{``....```^/&[(((((///^^^[&$@$$*//&/,.````,[`^\
11  [[[[[!((((||(((((|(/*&&$$&*/&oooo//`````````````o(^(|(((((^^^^^/o$$$&&&oo/-````([[`-\
12  ////[(((((|||!((!//&@@@@@@@#&$$oo&/`````````````/^^^((((|^^^^^^|[ooo&&&&ooo`````````-\
13  [([[[[[[!!!![[[/$$^_^^^|^(o$@@@&//``....`````/||(^^^(///^^^^|(/&oo*o&&/.````````^\
14  _;-:;-[([[(||!([$#^_______~^!^o$$/&`````````````([[[[///(((((/*/!$$&&/^^/o`````````~\
15  ~;-:~[[[[(!!((&@[///(^__,__^((@&[/`````````````(///(///$/*(^!&(((//[((^/(`````````^\
16  ;;~`,([[!!!![($-_(((o/{/{*|^(/$&//``....`(!/$**/&_^(//(/(^^~^|((/(_//(_,````````^\
17  //*&o((((((|(((^o*#*/^(&**/*&($o//`````````(/-_!((___^^(||||^!||((/|!^-^.````````-\
18  ((((((((!!!!|||!-^(^^^/o*&&&*//&{o`````````[o`|||-_:_|/((^^^^((((^^^-(^.````````^\
19  [[[[[(((||||!!^^((((^|{(^~!|([oo&&`````````(/`^([//*@*/[////{{{(||((/&o[o.````````-\
20  ////o!((||||(^(((/(^(&&{|(((//(//o&&`````````(/(([(((//((^(////{[([[[((*o`````````-\
21  o&{[/[[[!!!!((/(|^^|(((//&&/^/[&oo`````````````,///(//&/&&{(((////[((^;`````````^\
22  /[[[[(((||||!(((((((#/{/**/![$$&&``.`````````[///(/[[([////[/////(|[$.````````-\
23  /(/`!!((||||(((((|/**/[{//*/@$$$&&`````````````[[/*/[(((//**o&&///(/oo/````````-\
24  oooo*/[[!!!!(((/(^^^((/*//o$@@@@$$`````````````.!(((////&&&$*//(!/o&&&o/``````^\
25  &&&&$$$$*|||!(_[/***&&&&*/,##$$$@&`````````````````/*&$$$&&&*///{/&&&$$*o//(````-\
26  $$$$@@@@/!!![/^[(/&&@@&&o;_//$//$$``....````````````//*&@&&o/*//$&&$&&&&//////@\
27  $$$$@@@@/[!!*#,,![/&$$*_._$$&&@/$$`````````````(/o/o&@&$o&&o*###$$**&&&&////&&&@\
28  @@@@@@@@@&&&&$$-.,,^/[/;,:/##$$$&&&&@$#@@&o///*$&&&&$$@&&&&$$$#&&&$$$$&&&o*/$\
29  $@@@@@&&ooo$@@@:;[^^((&{-$$$@@@@@$$$$&&&&///&&@$&&$$@#@@$&&&&$@@$$$$$$&/oo/&&&@\
30  $$$&&&&&&$$$@@@_[/-[//_(/$$$@@@@$$$$$&&&///o&@@@$$$@$@**&&$##@@$$$&&&&o&&&&&@\
31  &&&&$$$$&@&&$$$-_^/^[;;-##&&$$$$@@@&$$$$&&&&$$$$&&@#@@&*o$#&$$$@@$$$$@#$&&&$$$$\
32  ",
```

七、 【技术点与创新点】

1. tab 补全

1) 指令补全:用户输入指令的一部分,按下 Tab 键,补全整条指令。

```
30      else
31      {
32          for (int i = 0; i < num_cmd; ++i)
33          {
34              const char* cmd_name = all_commands[i].name;
35              const char* pos = mismatch(cmd_name, cmd);
36              int len = pos - cmd_name;
37              if (*(cmd+len) == 0)
38              {
39                  puts(pos);
40                  strcpy(cmd+len, pos);
41                  break;
42              }
43          }
44      }
45  }
```

循环遍历所有的指令，如果现在输入的指令时某条指令的开头部分，就在屏幕上输出整条指令，并补充 cmd 字符串。all_commands 是在 command.h 中定义的数组，在操作系统 C 模块的 main 函数中添加，如下图：

```
8   typedef void(*funcType)(const char*);
9   typedef struct Command {
10      char name[MAX_CMD_LEN+1];   //指令名
11      funcType func;      //指令要做的事情
12  } Command;
13
14  Command all_commands[5];
15  int num_cmd = 0;
16
17  void addCommand(const char* cmd, funcType func)
18  {
19      strncpy(all_commands[num_cmd].name, cmd, MAX_CMD_LEN);
20      all_commands[num_cmd].func = func;
21      num_cmd++;
22  }
23
```

```
6   void main()
7   {
8       addCommand("ls", ls);
9       addCommand("info", info);
10      addCommand("clear", clear);
11      addCommand("./", run);
12      addCommand("shutdown", shutDown);
13
```

2) 程序名补全：用户输入./指令（执行程序），按下 Tab 键，自动补全对应程序名字。

```
12  void tabComplete(char* cmd)
13  {
14      if (starts_with(cmd, "./"))
15      {
16          cmd += 2;
17          for (int i = 0; i < num_program; ++i)
18          {
19              const char* process_name = all_programs[i].name;
20              const char* pos = mismatch(process_name, cmd);
21              int len = pos - process_name;
22              if (*(cmd+len) == 0)
23              {
24                  puts(pos);
25                  strcpy(cmd+len, pos);
26                  break;
27              }
28          }
29      }
```

循环遍历所有现有程序，如果现在输入的程序名（cmd 去掉./和可选空格）是某程序名的开头，就输出整个程序名，并补充 cmd。all_programs 是在 program.h 中定义的数组，在操作系统 C 模块的 main 函数中添加，如下图：

```
5   typedef struct Program {
6       char name[21];
7       int cylinder;
8       int track;
9       int start_sector;
10      int bytes;
11      int loadAddr;
12  } Program;
13
14  Program all_programs[5];
15  int num_program = 0;
16
17  void addProgram(const char* program, int startAddr, int endAddr, int loadAddr)
18  {
19      Program* newProgram = all_programs + num_program;
20      num_program++;
21      strncpy(newProgram->name, program, 20);
22      newProgram->bytes = endAddr - startAddr + 1;
23      int sector = startAddr / 0x200;         //物理扇区号
24      int y = sector / 18;
25      newProgram->cylinder = (y >> 1);
26      newProgram->track = (y & 1);
27      int z = sector % 18;
28      newProgram->start_sector = z + 1;
29      newProgram->loadAddr = loadAddr;
30  }
```

```
13      addProgram("sorry.com", 0x1800, 0x710b, 0x1000);
14      addProgram("rabbit.com", 0x7200, 0xc347, 0x1000);
```

这里有一个难点是如何从物理扇区映射到相对扇区，即柱面号、磁头号、起始扇区号的计算。

1.44M 的软盘结构如下：

A. 结构：2 面、80 道/面、18 扇区/道、512 字节/扇区
   扇区总数=2 面*80 道/面*18 扇区/道=2880 扇区
   存储容量= 512 字节/扇区*2880 扇区=1440 KB

B. 2 面：编号 0-1；
   80 道：编号 0-79；
   18 扇区：编号 1-18；

C. 相对扇区号：共 2880 个扇区，相对扇区号范围为 0-2879

扇区物理号	相对扇区号
0 面，0 道，1 扇区	0
0 面，0 道，2 扇区	1
0 面，0 道，3 扇区	2
……	
0 面，0 道，18 扇区	17
1 面，0 道，1 扇区	18
……	
1 面，0 道，18 扇区	35
0 面，1 道，1 扇区	36
0 面，1 道，18 扇区	53
1 面，1 道，1 扇区	54
……	

注意下红色字，它表明软盘的排列“*不是把 0 面先排完了再开始排 1 面，而是交替排列的*”。

例如：A=1 面，B= 15 道，C=7 扇区，这就是它的物理扇区号，现在进入关键问题——如何计算相对扇区呢？计算相对扇区时，参照上面的软盘结构排列表。我们应该清楚在 15 道之前，即 0～14 道里面，每道都有有 18 个扇区，而又 0、1 两面都有磁道，故而在 0～14 道有(0 道-14 道)*2 面*18， 在计算第 15 道时，注意下我们要计算的 15 道是在 1 面，而 1 面之前的 0 面 15 道，有 18 个扇区，而在 1 面的 15 道磁道中，有 7 个扇区。一共有 0 面的第 15 道 18 个扇区+1 面第 15 道 7 个扇区-1,所以上述例子中的相对扇区号是(0 道-14 道)*2 面*18+0 面的第 15 道 18 个扇区+1 面第 15 道 7 个扇区-1。

有了上面的叙述，物理扇区号除以 18 求出来商就是磁道数，而余数加 1（扇区从 1 开始编号）即是相对扇区数，由于除以了 18，那么前面的结构表就可以如下表示：

0 面，0 道
1 面，0 道
0 面，1 道
1 面，1 道
0 面，2 道
1 面，2 道
……
0 面，12 道
1 面，12 道
……

磁道除以 2 表示当前物理磁道号(因为是交替排列的！注意看上表的结构),而余数就可以表示磁头号。

2. 动图打印

了解视频原理的人都知道，视频其实是由非常多帧图片一帧一帧播放来实现的，动图也是如此，只不过帧数不如视频。因此只要把动图分解成一帧一帧，然后将每一帧转成字符串，然后按照一定的时

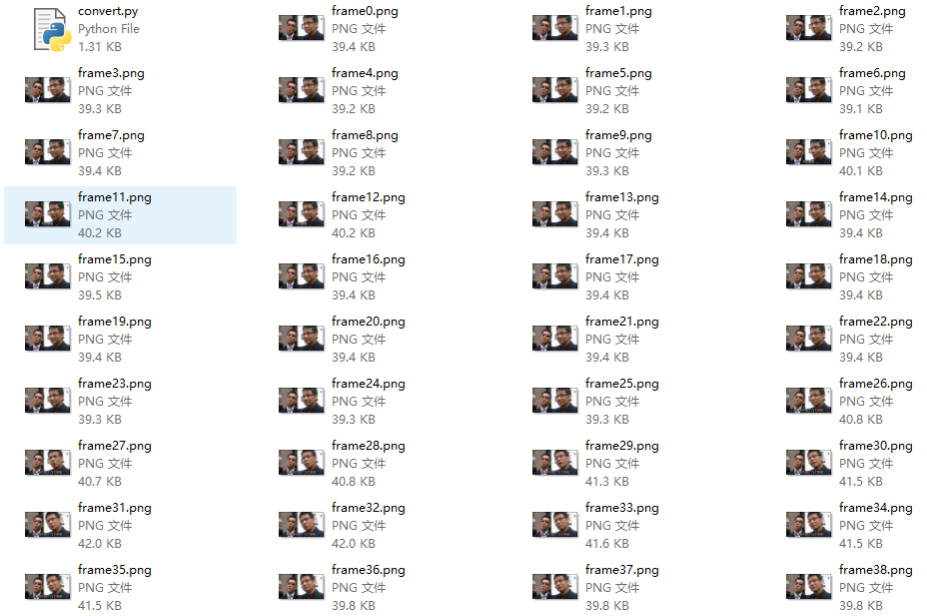
间间隔输出就达到了“动”的效果。问题的关键是，如何分解动图，以及如何把一张静态图转成字符串？Python 确实很强大，它的 opencv 和 numpy 库为我们提供了图片处理的功能。分解和转字符串的代码如下：

```
frames = []
def get_frame_count(videofile):
    frame_count = 0
    video_cap = cv2.VideoCapture(videofile)
    while (video_cap.isOpened()):
        ret, frame = video_cap.read()
        if ret is False:
            break
        frame_count += 1
        frames.append(frame)
    return frame_count
```

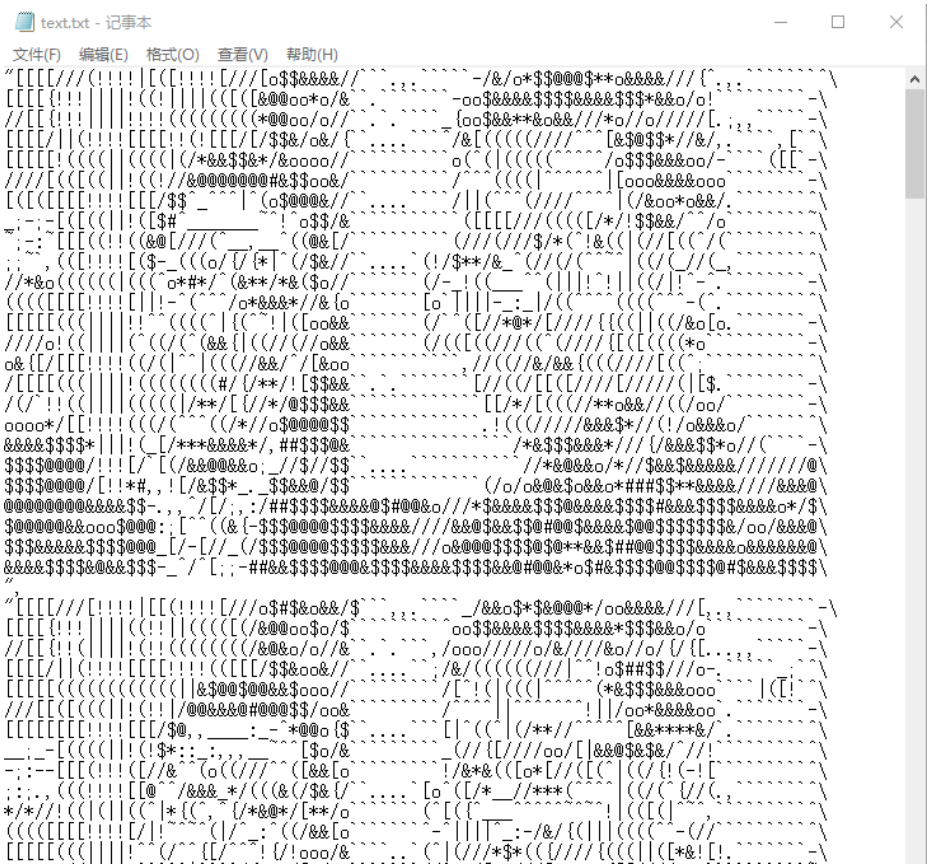
```
def video_to_frame(jiange):
    text = open("text.txt", "w")
    i = 0
    for frame in frames:
        if i % jiange == 0:
            gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
            cv2.imwrite("frame.png", frame)
            frame_to_char(text)
        i += 1
    text.close()
    os.system("del frame.png")
```

```
def frame_to_char(text):
    gray_char=['@','#','$','&','*','o','/','{','[','(',
               '|','!','^','~','-','_',':',';',',','.','`',' ']

    frame = resize_frame('frame.png')
    text.write('\"')
    for y in range(size[1]):
        for x in range(size[0]):
            gray = frame.getpixel((x,y))
            char = gray_char[int(gray/(255/(len(gray_char)-1)))]
            text.write(char)
        text.write("\\\n")
    text.write('",\n')
```

video_to_frame 函数实现动图分解功能，frame_to_char 函数实现把一张静态图转成字符串。如果在 video_to_frame 函数中把每张图都输出（即 cv2.imwrite("frame%d.png" % i, frame)，并去掉 os.system("del frame.png")），就能看到图片被分解之后每一帧，如下图（总 167 帧）：



转好的字符串在"text.txt"文件中，如下图：



八、 【实验总结】

这次实验历时两周，第一周用的是 TCC+TASM。这套工具的好处是 TASM 编译出的代码就是 16 位的，能与 TASM 汇编代码完美结合，因此就不用考虑在底层 C 与汇编相互之间传递参数和返回的一系列令人头疼的问题。但是，

TASM 汇编的语法实在是令人抓狂。首先，TASM 的入口地址只能是 100h，因此为了能让正确地把控制权交给用户程序，caller 就要用基地址:偏移地址的方式得到用户程序的物理地址。这和 NASM 比起来真的很不方便，因为 NASM 是任意入口的，org 可以是任意合理的值，caller 直接跳到用户程序 org 的地址就可以了。其次，TASM 的语法实在是很冗余。比如，为了获取某个地址处的值，需要加 PTR，为了获取某个变量，需要交 OFFSET。而在 NASM 中，不用中括号括住的变量就是变量本身的值，用中括号括住表示变量（被当做地址）处的值，多简洁！

在 TASM 中折腾了快一周的时候，老师突然公布消息，GCC 和 NASM 工具的使用走通了！这真的是一个天大的好消息，真的太喜欢 NASM 简洁方便的特性了！于是，我马上开始探索如何使用 GCC+NASM 进行混合编程。其中遇到了很多的挫折，比如一开始用 objcopy 产生的二进制文件各个段之间相隔很远，以至于出现大片大片的 0，文件很大，后来通过在网上查看各种各样的文档，以及实验参考文献的帮助下，发现—oformat 参数可以解决这个问题；GCC 产生的 32 位代码和 NASM 的 16 位汇编代码之间函数调用如何传参，参数多少位，如何正确返回，这些都是需要仔细斟酌和耐心探索的技术要点和难点。虽然过程很曲折，但是这次实验给我的收获比前两次实验的多得多，一个可能是时间比较长的原因，但更重要的是整个的探索过程让我对 GCC 编译链接过程、编译选项以及函数调用机制都有了切身的体会。

完成了混合编程的探索，我开始做一些比较有趣的事情——打印动图！把动图以字符串的形式输出的做法在很久之前就在网上流行起来了，但是很多教程都是用比较高级的语言，比如 Python。如何在 C 或者汇编完成这件事？这确实是一个比较大胆而且富有挑战性的想法！仔细研究了网上的一些教程，对动图的原理和动图转字符串的方法有了大致了解之后，我就开始动手了。把这么多帧的字符串放到程序中，难免使得程序占用很大的内存空间，通过合理地控制帧数和程序加载的位置，可以让动图输出了。但是期间也遇到一些问题，比如设计两个动图程序，第 2 个程序（在磁盘相对比较后的扇区）的显示就不正常了。后来不断 debug 才发现，是因为调用读磁盘中断的时候柱面号和磁头号不正确，因为之前都是默认 0，而这次的用户程序比较大（一个 40 几个扇区），而一个磁道只有 18 个扇区，默认为 0 肯定是不对的。后来在网上寻找答案，了解磁盘分布以及柱面、磁道、扇区之间的关系，得到计算这些量的公式，才解决了读磁盘错误的问题。这同时也让我对磁盘有了更深的了解。

九、 代码清单

```
project3
    --gcc+nasm
        --bin
    --include
        --crt0.inc
        --klib.inc
        --ctype.h
        --stdio.h
```

    --stdlib.h
    --string.h
--lib
    --crt0.o
    --klib.o
--video2char
    --convert.py
--操作系统
    --os_a.asm
    --terminal.c
    --terminal.h
    --command.h
    --program.h
--软盘文件
--引导程序
    --boot.asm
--用户程序
    --rabbit.c
    --sorry.c

十、 【参考文献】

1. 实验参考文献/蔡日俊 GCC+NASM 环境.pdf
2. 软盘结构(磁头号和起始扇区的计算方法)
   https://blog.csdn.net/littlehedgehog/article/details/2147361