# 实验三实验报告

## 【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

## 【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；

## 【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。

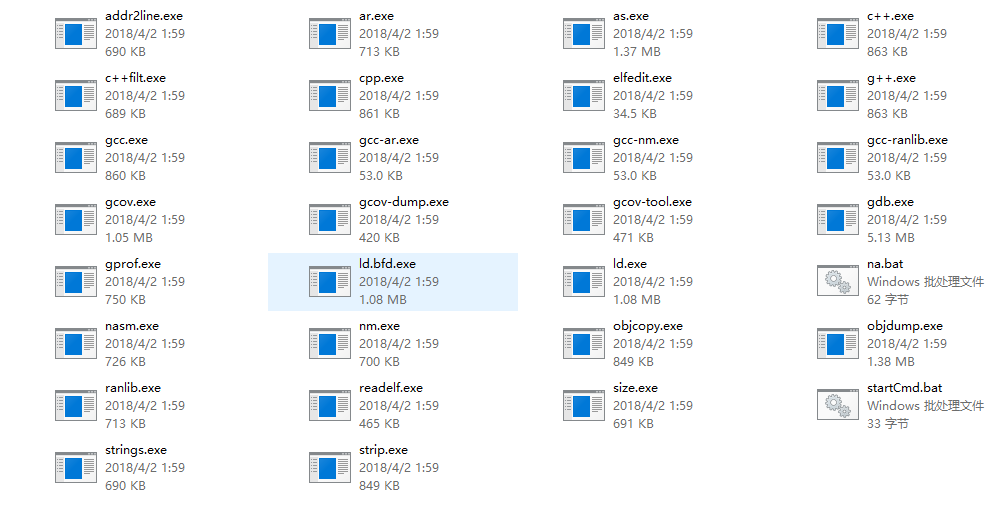
## 【实验要求】

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。

## 【实验方案】

1. **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

* 1. NASM汇编命令

1. 生成com文件

nasm -f bin <afile>.asm -o <afile>.com

1. 生成链接文件

nasm -f elf32 <afile>.asm -o <afile>.o

* 1. 链接命令

1. 生成引导扇区程序：

ld -N <afile>.o <cfile>.o -Ttext 0x7c00 --oformat

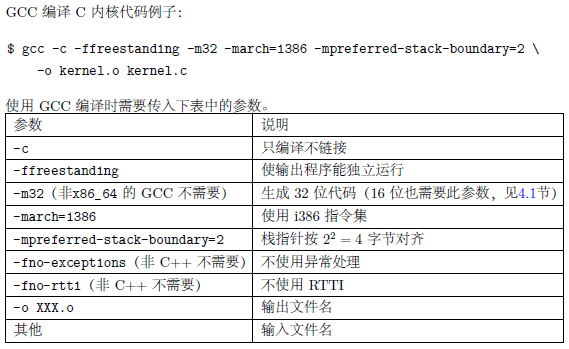
binary -o <output>.bin

1. 生成COM程序：

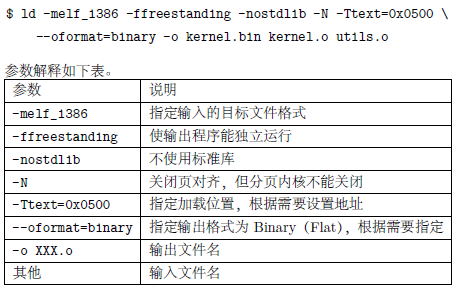
ld -N <afile>.o <afile>.o -Ttext 0x100 --oformat

binary -o <output>.com

1. **实验原理**
   1. **编译参数**



* 1. **链接参数**

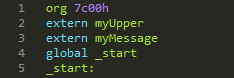


* 1. **GCC与NASM交叉编译**
     1. 汇编模块中调用C模块中的函数
     + 调用前要用extern声明C模块的函数
     + 根据C中函数原型，用栈传递参数，顺序后参先进栈
     + 调用C函数后，要将栈中参数弹出
     + 进栈出栈以4字节为单位
     + 返回地址为4个字节，因此除了传递参数，在call之前必须把cs寄存器的值入栈（cs+ip总共4个字节）
     1. C模块中调用汇编函数和引用变量
        + 汇编模块的过程从栈中取得参数，不必出栈，直接引用栈中的值，顺序与C进栈对应
        + 由于NASM的ret指令只取栈顶两个字节（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

## 【实验过程】

1. 命令行选项与新汇编语句理解
2. -Ttext addr

以elf32模式生成汇编.o文件时，org指令被禁用，如下图：

 C:\Users\96552\AppData\Roaming\Tencent\Users\965524991\TIM\WinTemp\RichOle\8B7Q(2N[QJW)ZD$4M8TRN6X.png

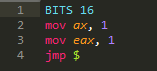
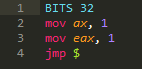
链接时通过-Ttextaddr的方法指定入口地址，即-Ttext是链接时将初始地址重定向为0x7c00（若不注明此，则程序的起始地址为0）。

1. 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。

1. 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:

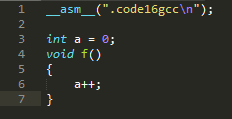
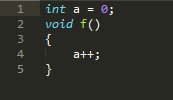
C:\Users\96552\AppData\Roaming\Tencent\Users\965524991\TIM\WinTemp\RichOle\A@{[17@@C00B@@FYEMUT1N3.png

C:\Users\96552\AppData\Roaming\Tencent\Users\965524991\TIM\WinTemp\RichOle\81646ZK(SJ2GFXWJ1)@H%8X.png

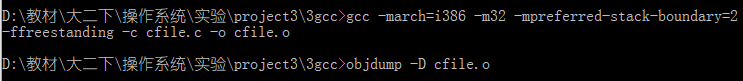
当NASM在BITS 16状态下时，使用32位数据的指令要加一个字节的前缀0x66，要使用32位的地址，则加上0x67前缀。在BITS 32状态下，相反的情况成立，32位指令不需要前缀，而使用16位数据的指令需要0x66前缀，使用16位地址的指令需要0x67前缀。

1. \_\_asm\_\_(".code16gcc\n")

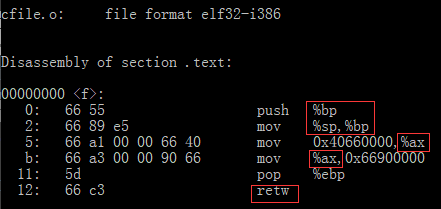
C语言程序的开头使用了\_\_asm\_\_(".code16gcc\n") 嵌入汇编指令，以指示 as 生成 16 位代码。有什么区别？编写一个简单程序来测试：

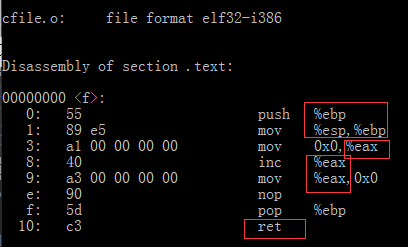
 

分别用objdump反汇编生成的.o文件，如下：



加嵌入汇编和不加的结果如下：





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。

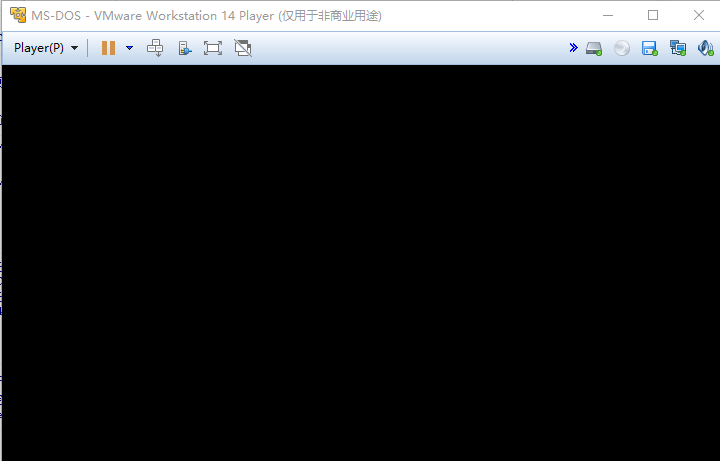
另外，调用gcc时除了指定-c选项指示它只编译不连接外，还要指定-m32选项，这样才会生成32位的汇编代码，而只有在32位的汇编代码中使用.code16gcc指令，才能编译成 16 位的机器码。如果没有指定-m32选项，则生成的是64位汇编代码，然后汇编时会出错。使用-m32选项后，生成的目标文件是 ELF32 格式。ELF32 格式的目标文件只能和ELF32格式的目标文件连接，这也是为什么前面的nasm和ld需要指定elf32和-m elf\_i386选项。

1. C与汇编之间相互调用的探索
   1. 返回地址
      1. 汇编调用C

有几行代码让人感到疑惑：



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。

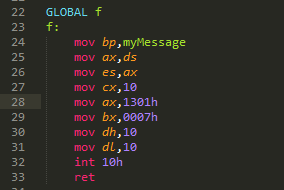
* + 1. C调用汇编

测试代码如下：

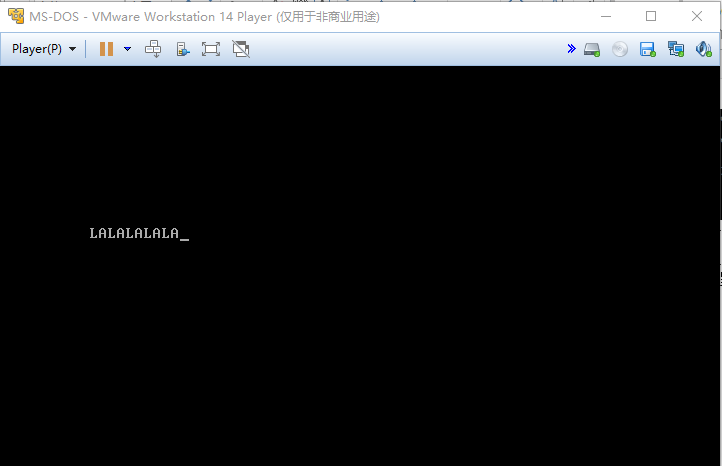
afile.asm



cfile.c

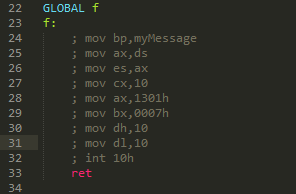


打开虚拟机结果正常：

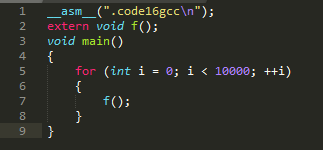


可是真的没有问题吗？不妨多调用几次f函数。

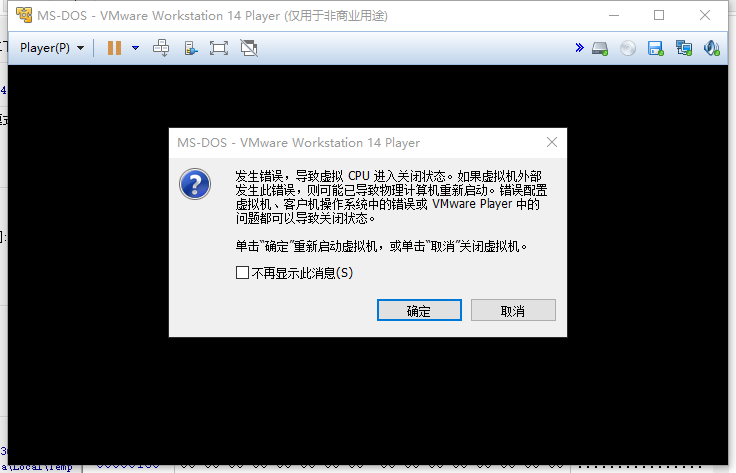
afile.asm更改如下：



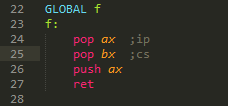
cfile.c更改如下：



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：



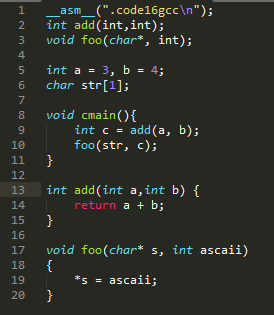
如此再次运行就没有问题了。可是这样做的缺点是需要改动ax和bx，我们并不想在一个函数中更改不必要的寄存器。

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。

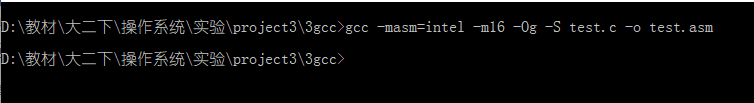
C:\Users\96552\AppData\Roaming\Tencent\Users\965524991\TIM\WinTemp\RichOle\{QPZ1D1QDV_GL]@2Q_`S3L7.png

* 1. 参数传递

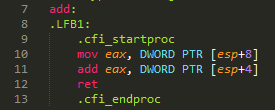
编写一个简单的C程序，观察GCC编译产生的AT&T汇编代码（局部），代码如下：

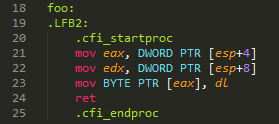


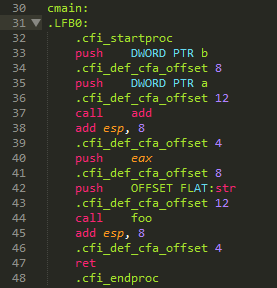
编译命令：



产生的汇编文件代码如下：



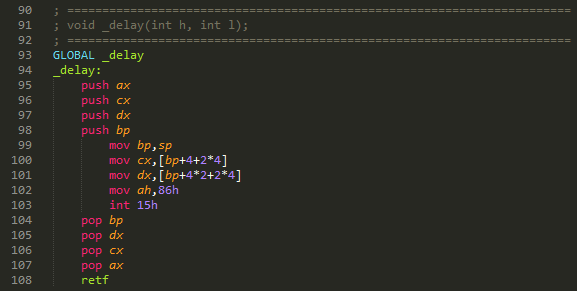




观察其中一些高亮的代码，可以发现一个规律：

* + - 1. C语言程序中，GCC编译后，变量名和C中一样，函数名也如此；
      2. 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：



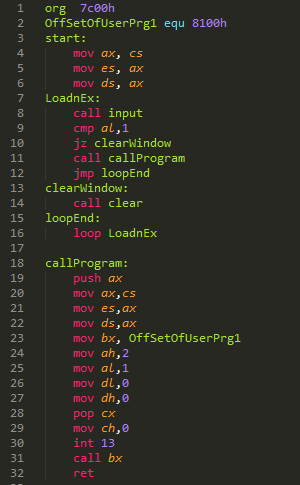
这是一段延时代码，延迟cx:dx微秒。2\*4是因为把ax、cx、dx、bp入栈，每个2个字节。取cx时+4是因为h传进来的时候是4个字节，加了4才能让bp指向h的开始地址；取dx时加4\*2是因为l在h之下4个字节的地方开始。

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。

1. 从监控程序中分离操作系统内核

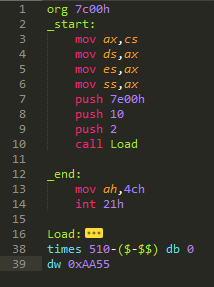
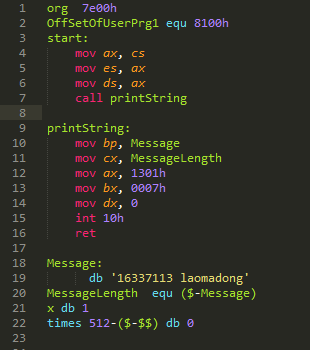
在实验二中，监控程序包含了引导程序和操作系统。为了把操作系统内核分离出来，只需要把操作系统作为一个用户程序，在引导程序中加载执行操作系统，并把控制权交给它。

实验二监控程序部分代码如下：

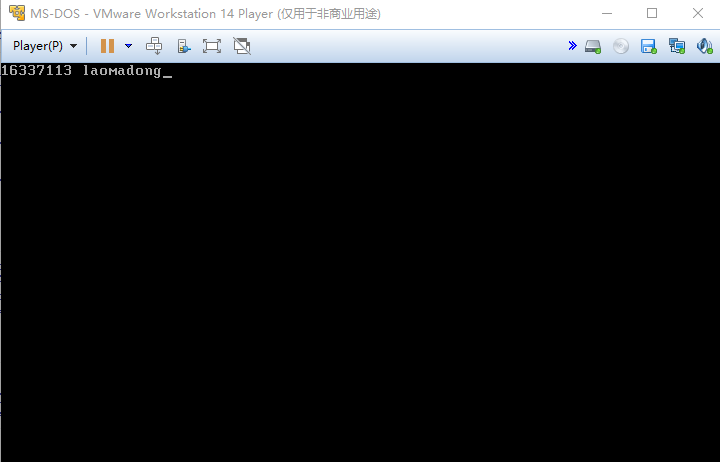


其中，起到引导程序作用的代码只是org 7c00h语句，它负责把代码加载到内存7c00h的地方，使得开机的时候系统跳到7c00h地址的时候可以正确执行以上代码。其余代码都是操作系统的部分，因此把以上代码分成两部分。

引导程序（局部）： 操作系统（局部）：

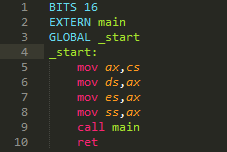
操作系统放在从第二个扇区开始的10个扇区中，加载到内存的7e00h处。引导程序的作用就是，从磁盘中读操作系统并加载到7e00h，然后call到7e00h的地方，就把控制权交给了操作系统。操作系统的org是7e00h，要做的事情就是打印个人信息。不妨测试一下！



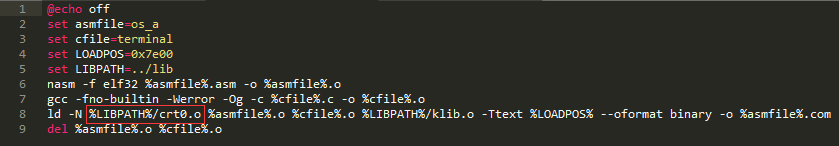
测试结果符合预期，操作系统已经被分离出来了。之后扩展操作系统代码，和C交叉编译，让它加载并执行用户程序，就完成了一个操作系统内核原型。最终做出的操作系统代码详见源代码文件。

1. 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：



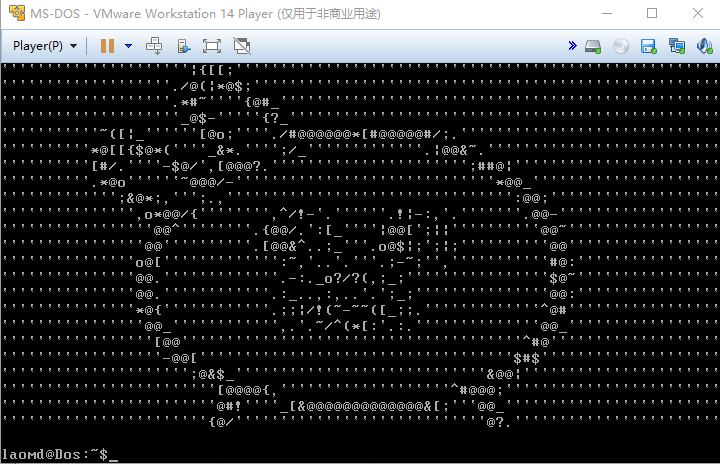
代码的作用是初始化一些段寄存器，然后调用C中的main函数，并在main函数返回时回到上一层调用者。通过查阅一些GCC文档，发现C程序在main()执行之前还有启动代码，通常叫作c0、crt0之类，.exe真正的入口点在这里面，它执行一系列的初始化工作再调用main()。于是就把上面这段代码当做启动代码，编译成一个.o文件，所有需要交叉编译的程序只需要链接这个.o就可以。如下：



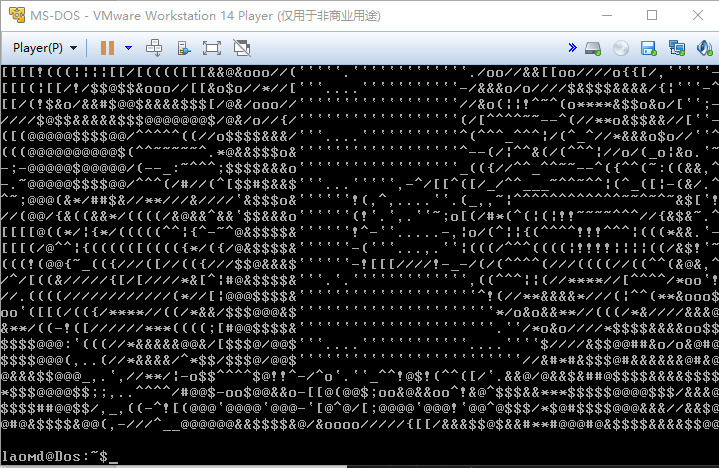
1. 编写用户程序

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：

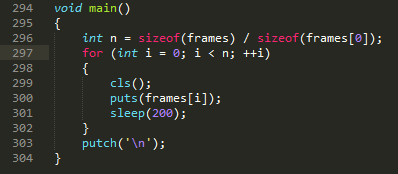
兔子动图：



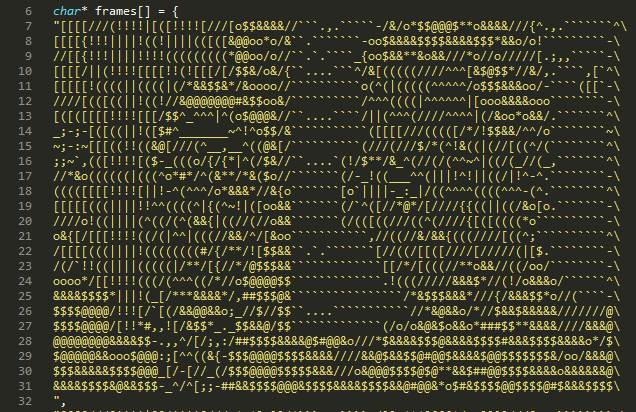
Sorry动图:

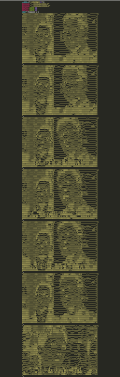
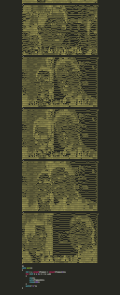


方法是把动图的每一帧转换成字符串，然后建立一个很大的字符串数组，每隔一段时间播放一帧对应的字符串。



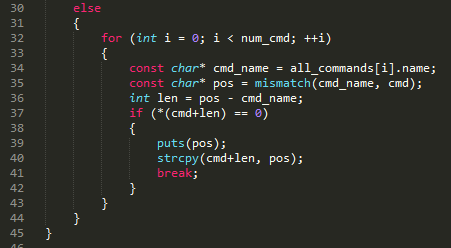
其中frames是字符串数组，部分内容截图如下：



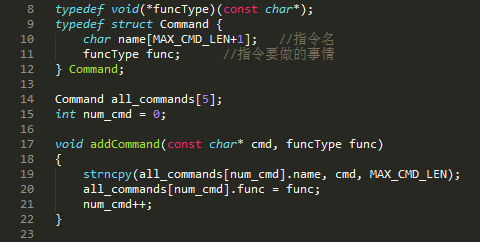
## 【技术点与创新点】

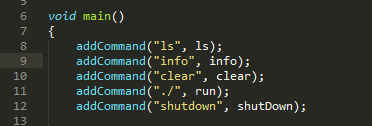
1. tab补全
   1. 指令补全：用户输入指令的一部分，按下Tab键，补全整条指令。



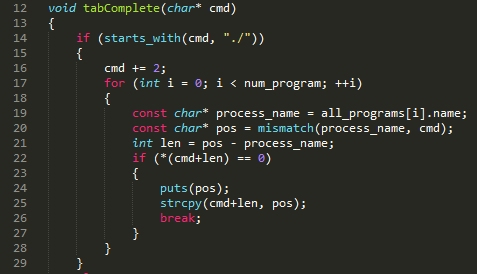
循环遍历所有的指令，如果现在输入的指令时某条指令的开头部分，就在屏幕上输出整条指令，并补充cmd字符串。

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：



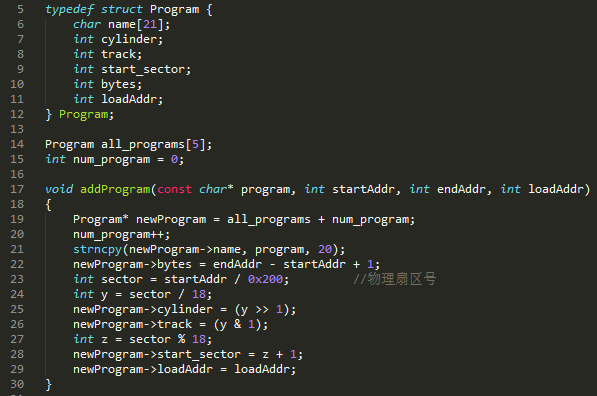


* 1. 程序名补全：用户输入./指令（执行程序），按下Tab键，自动补全对应程序名字。



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：



C:\Users\96552\AppData\Roaming\Tencent\Users\965524991\TIM\WinTemp\RichOle\$G5$Q)RGT6CT%N%N({FC)ZQ.png

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：

1. 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

1. 2面：编号0-1；

80道：编号0-79；

18扇区：编号1-18；

1. 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

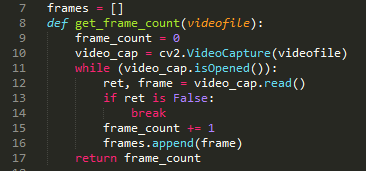
1面，12道

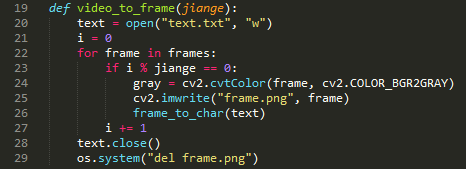
.......

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。

1. 动图打印

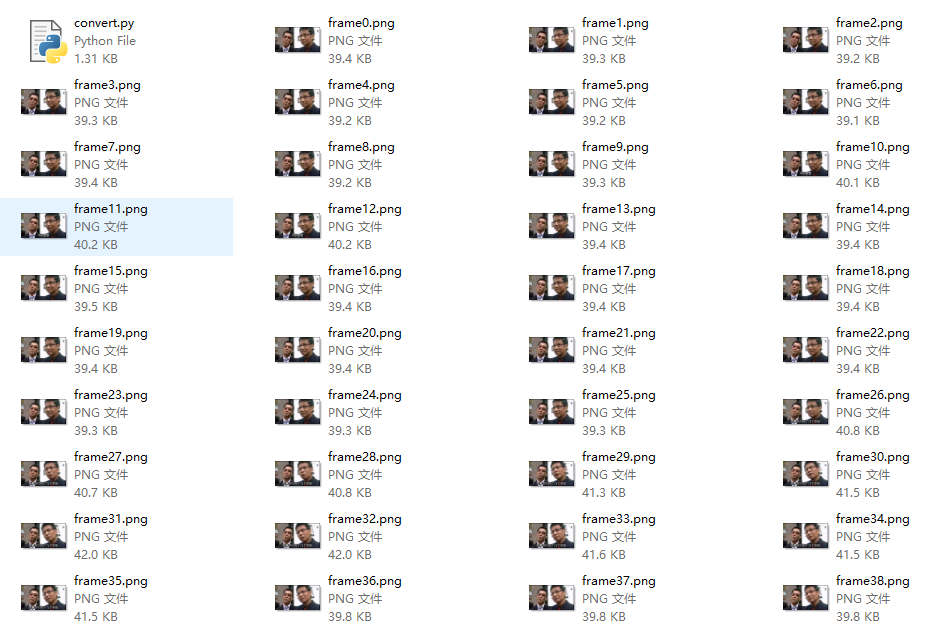
了解视频原理的人都知道，视频其实是由非常多帧图片一帧一帧播放来实现的，动图也是如此，只不过帧数不如视频。因此只要把动图分解成一帧一帧，然后将每一帧转成字符串，然后按照一定的时间间隔输出就达到了“动”的效果。问题的关键是，如何分解动图，以及如何把一张静态图转成字符串？Python确实很强大，它的opencv和numpy库为我们提供了图片处理的功能。分解和转字符串的代码如下：



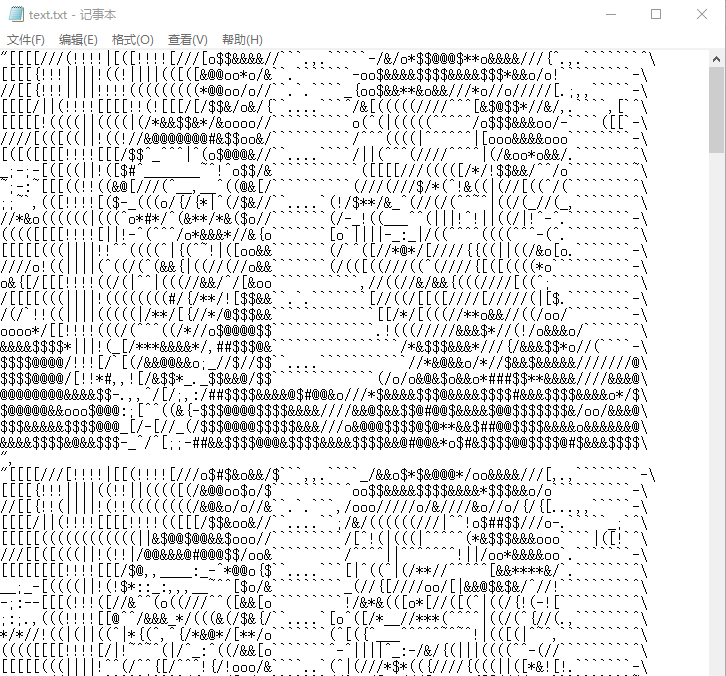




video\_to\_frame函数实现动图分解功能，frame\_to\_char函数实现把一张静态图转成字符串。如果在video\_to\_frame函数中把每张图都输出（即cv2.imwrite("frame%d.png" % i, frame)，并去掉os.system("del frame.png")），就能看到图片被分解之后每一帧，如下图（总167帧）：



转好的字符串在"text.txt"文件中，如下图：



## 【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。

## 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

## 【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