

Report

18340161 田蕊 18级数据科学与计算机学院 超算方向 田蕊

一、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

二、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

三、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（函数返回），汇编模块显示统计结果。执行程序可以在DOS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

四、试验方案

（一）、实验工具

编译：TCC+TLINK+TASM

软件：dosbox, virtual box, notepad++

（二）、实验方案

分析C语言编译出的汇编代码

编写好C样版程序后用TCC进行编译得到汇编语言程序后进行代码分析。

五、实验过程

(一)、分析C语言编译出的汇编代码

实验步骤

1. 书写样版C程序
2. 编译得到.asm文件
3. 分析汇编语言文件中的全局变量，局部变量，变量初始化，函数调用，参数传递情况

样版C程序

由于老师没有固定样版C程序，所以我们要先自己写一个样版C程序来进行分析。样版C程序普遍十分简单，所以我们也只实现一个简单的函数，函数的功能是选出两个数中较大的那一个。下面给出样版C程序的代码

```
int f(int a,int b){
    int ret = -1;
    if(a>b) ret = a;
    else ret = b;
    return ret;
}

int main(){
    int temp[2] = {0};
    int res = 0;
    temp[0] = 100;
    temp[1] = 50;
    res = f(temp[0],temp[1]);
    return 0;
}
```

程序十分简单，但是要注意一定要严格按照C语言格式来书写，不可以有C++风格的代码。

接下来在DOSBOX中执行这条指令：tcc -S filename.c

这里要注意-S一定要大写，否则TCC会报错：incorrect command line option: -s

正确输入指令后结果如下所示

```
A:\ASSEMBLY\OSEX3\MYCODE1>tcc -S ques1.c
Turbo C Version 2.01 Copyright (c) 1987, 1988 Borland International
ques1.c:
Warning ques1.c 15: 'res' is assigned a value which is never used in function ma
in

Available memory 458698
```

之后就可以看见在同一目录下多了一个同名的.asm文件。打开文件后可以看见汇编代码，如下所示

```
ifndef ??version
?debug macro
```

```

    endm
    endif
    ?debug  S "ques1.c"

_TEXT    segment byte public 'CODE'
DGROUP  group  _DATA,_BSS
    assume cs:_TEXT,ds:DGROUP,ss:DGROUP
_TEXT    ends

;数据段必须定义成这样,定义类型为word,组合类型为public,段类别为DATA
_DATA    segment word public 'DATA'
d@ label  byte
d@w label  word
_DATA    ends

_BSS     segment word public 'BSS'
b@ label  byte
b@w label  word
    ?debug  C E93044AB500771756573312E63
_BSS     ends

_DATA    segment word public 'DATA'
_global label  word
    dw 0
_DATA    ends

_TEXT    segment byte public 'CODE'
;    ?debug  L 3
_f proc  near
    push    bp
    mov bp,sp
;    ?debug  L 4
    mov ax,word ptr [bp+4]
    cmp ax,word ptr [bp+6]
    jle @2
    mov ax,word ptr [bp+4]
    mov word ptr DGROUP:_global,ax
    jmp short @3
@2:
;    ?debug  L 5
    mov ax,word ptr [bp+6]
    mov word ptr DGROUP:_global,ax
@3:
;    ?debug  L 6
    mov ax,word ptr DGROUP:_global
    jmp short @1
@1:
;    ?debug  L 7
    pop bp
    ret
_f endp
_TEXT    ends

_DATA    segment word public 'DATA'
    dw 0
    db 2 dup (0)
_DATA    ends

```

```

_TEXT    segment byte public 'CODE'
;  ?debug L 9
_main    proc    near
    push    bp
    mov bp,sp
    sub sp,4
    push    si
    push    ss
    lea ax,word ptr [bp-4]
    push    ax
    push    ds
    mov ax,offset DGROUP:d@+2
    push    ax
    mov cx,4
    ;执行了另一个函数作用是把数据段中的源串拷到栈中
    call    far ptr SCOPY@
;  ?debug L 11
    xor si,si
;  ?debug L 12
    mov word ptr [bp-4],100
;  ?debug L 13
    mov word ptr [bp-2],50
;  ?debug L 14
    push    word ptr [bp-2]
    push    word ptr [bp-4]
    call    near ptr _f
    pop cx
    pop cx
    mov si,ax
;  ?debug L 15
    xor ax,ax
    jmp short @4
@4:
;  ?debug L 16
    pop si
    mov sp,bp
    pop bp
    ret
_main    endp
_TEXT    ends

    ?debug C E9
_DATA    segment word public 'DATA'
s@ label byte
_DATA    ends

    extrn    SCOPY@:far
_TEXT    segment byte public 'CODE'
_TEXT    ends
    public _global
    public _main
    public _f
    end

```

分析

基本知识

由伪指令 `segment`和 `ends`围起来的部分，是给构成程序的命令和数据的集合体上加一个名字而得到的，称为段定义。在这个程序中，段定义指的是命令和数据等程序的集合体的意思，一个程序由多个段定义构成。

`group` 这个伪指令表示的是将 `_BSS`和`_DATA` 这两个段定义汇总名为 `DGROUP` 的组。

`_f proc` 和 `_f endp`围起来的部分，以及`_main`和`_main endp`围起来的部分，分别表示函数`f`和函数`main`的范围。编译后函数名前附带下划线是TCC的规定，在C语言中编写的`f`函数，在内部实际上是以`_f`这个名称处理的。

伪指令 `proc` 和 `endp` 围起来的部分，表示的是 过程(procedure) 的范围。在汇编语言中，这种相当于C语言的函数的形式称为过程。

末尾的 `end` 伪指令，表示的是源代码的结束。

全局变量及其初始化

在这个样版C程序中定义了一个`int`型的全局变量`global`并初始化为零，可以看到汇编的到的代码中有这样的一段程序

```
_DATA    segment word public 'DATA'
_global  label    word
        dw  0
_DATA    ends
```

这里定义了一个全局变量，因为`int`在内存中占四个字节就是一个`word`，`dw 0`的含义就是初始化为0。

局部变量及其初始化

在函数`f`中没有定义局部变量，但是在`main`函数中有定义一个名为`res`的`int`型局部变量，还有定义一个名为`temp`的`int`型数组，数组大小为2。观察汇编代码段，发现在`main`函数段之前，有这样的一段代码

```
_DATA    segment word public 'DATA'
        dw  0
        db  2 dup (0)
_DATA    ends
```

这里定义了两个变量，就分别是我们上面提到的`res`和`temp`变量。由于我们在C代码中将这两个变量都初始化为0，所以在汇编程序中也是 `dw 0` 和 `db 2 dup (0)`。

函数调用和参数传递

在主函数中我们调用了函数`f`，具体的调用代码部分如下所示

```
    mov word ptr [bp-4],100
;    ?debug  L 13
    mov word ptr [bp-2],50
;    ?debug  L 14
    push  word ptr [bp-2]
    push  word ptr [bp-4]
    call  near ptr _f
    pop  cx
    pop  cx
```

首先两个mov语句是对变量进行初始化，接下来的push语句是将变量压栈，进行参数传递，注意压栈的顺序是和函数参数表的顺序是反着的，也就是说参数表中后面的参数要先压栈。在这些都处理好之后就是调用函数，我们看见 `call near ptr _f` 就是调用了函数f，在调用结束后还要把参数弹栈，注意这一步是非常重要的，入果没有这一步会造成极大的错误。

（二）、一个简单的汇编程序和C程序混合编程

实验步骤

- 设计程序的函数及其接口
- 编写程序
- 编译链接结果

程序函数及其接口

C语言部分

```
int count(char* a,int b){
    int i = 0;
    int cnt = 0;
    for(i=0;i<b;i++){
        if(a[i]=='a') cnt++;
    }
    return cnt;
}
```

程序设计的要求是编写一个程序，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（函数返回），汇编模块显示统计结果。执行程序可以在DOS中运行。所以C语言需要实现的功能非常简单，只要简单的统计字符出现的次数即可，我们这里就统计小写字母'a'的出现次数。

汇编语言部分

相对于C语言来说汇编语言要复杂很多，我们这里简单地介绍一下

定义数据段

```
_DATA segment word public 'DATA'
    assume ds:_DATA
    tip1 db  "the message is:  bbaaazzssssaa", 0AH, 0DH, '$'
    tip1len equ $-tip1
    tip2 db  "the number of 'a' in the message is : ";38
    tip2len equ $-tip2
    mess db  "bbaaazzssssaa"
    len equ  $-mess

_DATA ends
```

首先要定义一下我们需要用到的数据，分别有提示语句和需要统计的字符串。

接下来程序的第一部分我们先输出提示语句，采用调用中断的方法，和前面的程序用到的一样，我们这里不赘述。但是值得注意的是在输出字符串的时候，一定要写成 `mov bp, offset DGROUP:tip1, DGROUP` 一定要加上，否则没有办法正确输出字符串。在提示语句都输出之后，我们将参数压栈后调用函数。

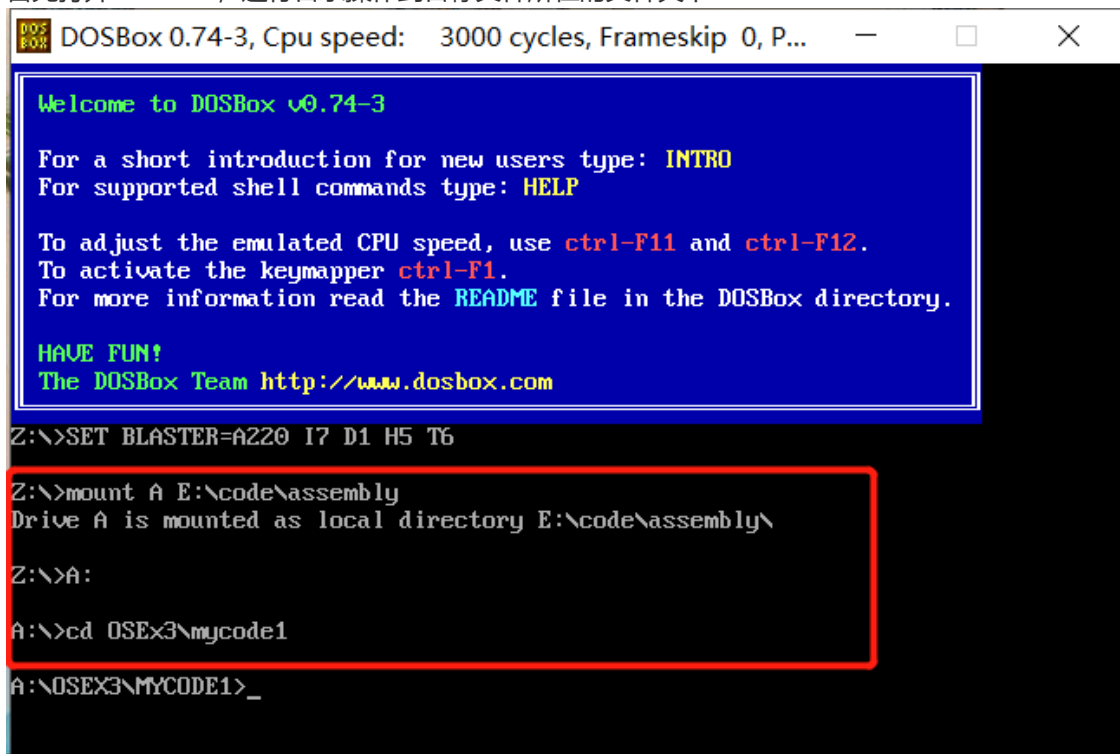
调用函数

```
mov ax, len
push ax
mov ax, OFFSET DGROUP:mess
push ax
call near ptr _count
pop cx
pop cx
```

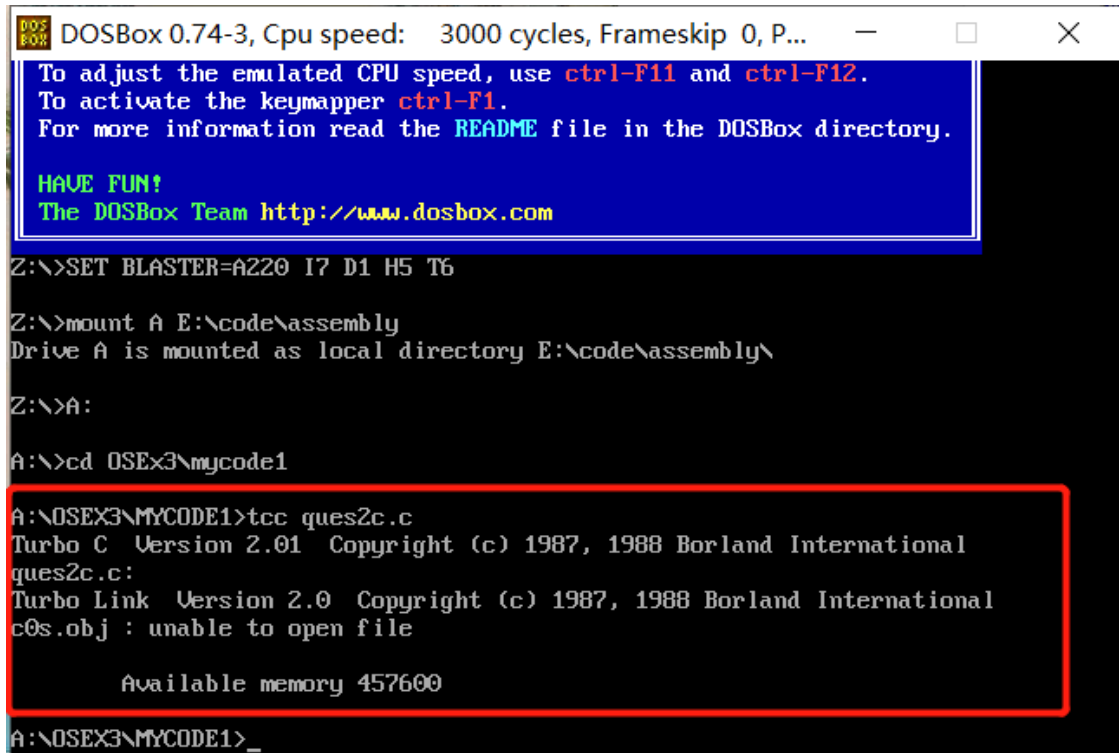
压栈的时候要记得倒着压，函数调用结束要记得 `pop cx`，有几个参数就要 `pop` 几次。调用函数后返回结果默认保存在 `ax` 寄存器中，因为只有一个字符所以在输出前要加上 48，也就是字符 '0' 的 ASCII 码，最后输出统计结果。

操作过程

1. 首先打开 DOSBOX，进行目录操作到目标文件所在的文件夹下



2. 之后对C语言文件进行编译



DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, P...

To adjust the emulated CPU speed, use **ctrl-F11** and **ctrl-F12**.
To activate the keymapper **ctrl-F1**.
For more information read the **README** file in the DOSBox directory.

HAVE FUN!
The DOSBox Team <http://www.dosbox.com>

```
Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>mount A E:\code\assembly
Drive A is mounted as local directory E:\code\assembly\

Z:\>A:

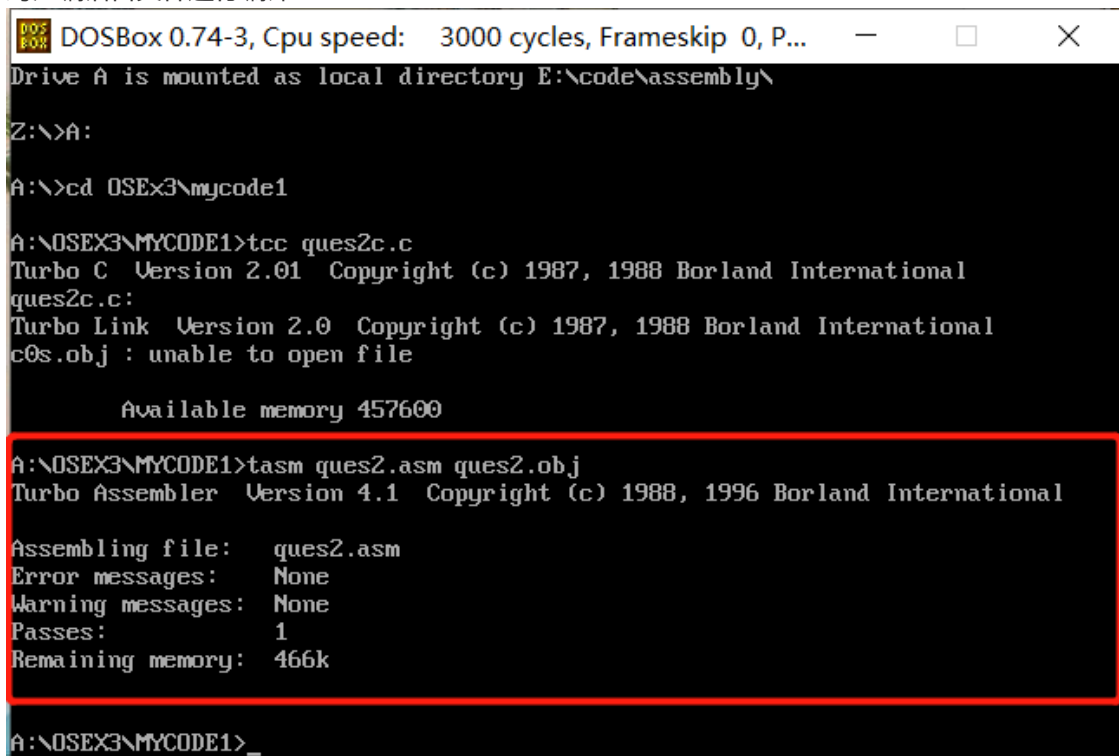
A:\>cd OSEX3\mycode1

A:\OSEX3\MYCODE1>tcc ques2c.c
Turbo C Version 2.01 Copyright (c) 1987, 1988 Borland International
ques2c.c:
Turbo Link Version 2.0 Copyright (c) 1987, 1988 Borland International
c0s.obj : unable to open file

Available memory 457600

A:\OSEX3\MYCODE1>_
```

3. 对汇编语言文件进行编译



DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, P...

Drive A is mounted as local directory E:\code\assembly\

```
Z:\>A:

A:\>cd OSEX3\mycode1

A:\OSEX3\MYCODE1>tcc ques2c.c
Turbo C Version 2.01 Copyright (c) 1987, 1988 Borland International
ques2c.c:
Turbo Link Version 2.0 Copyright (c) 1987, 1988 Borland International
c0s.obj : unable to open file

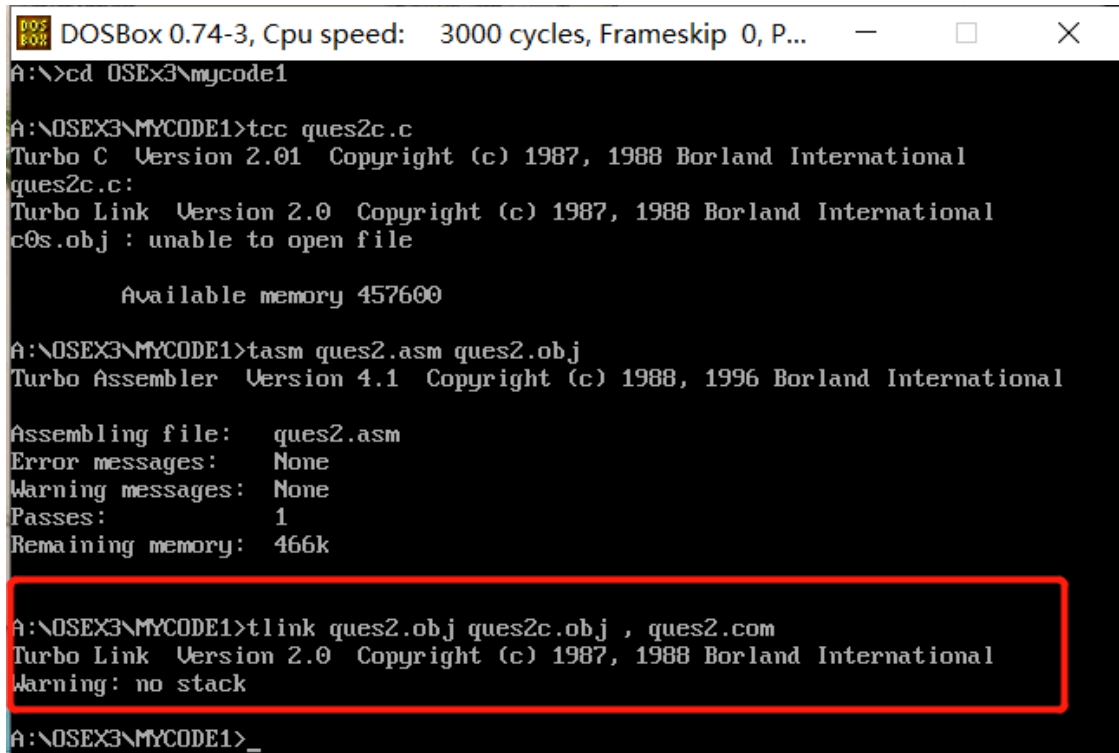
Available memory 457600

A:\OSEX3\MYCODE1>tasm ques2.asm ques2.obj
Turbo Assembler Version 4.1 Copyright (c) 1988, 1996 Borland International

Assembling file: ques2.asm
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 466k

A:\OSEX3\MYCODE1>_
```


4. 对编译好的两个文件进行链接



```
DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, P...
A:\>cd OSEX3\mycode1
A:\OSEX3\MYCODE1>tcc ques2c.c
Turbo C Version 2.01 Copyright (c) 1987, 1988 Borland International
ques2c.c:
Turbo Link Version 2.0 Copyright (c) 1987, 1988 Borland International
c0s.obj : unable to open file

Available memory 457600

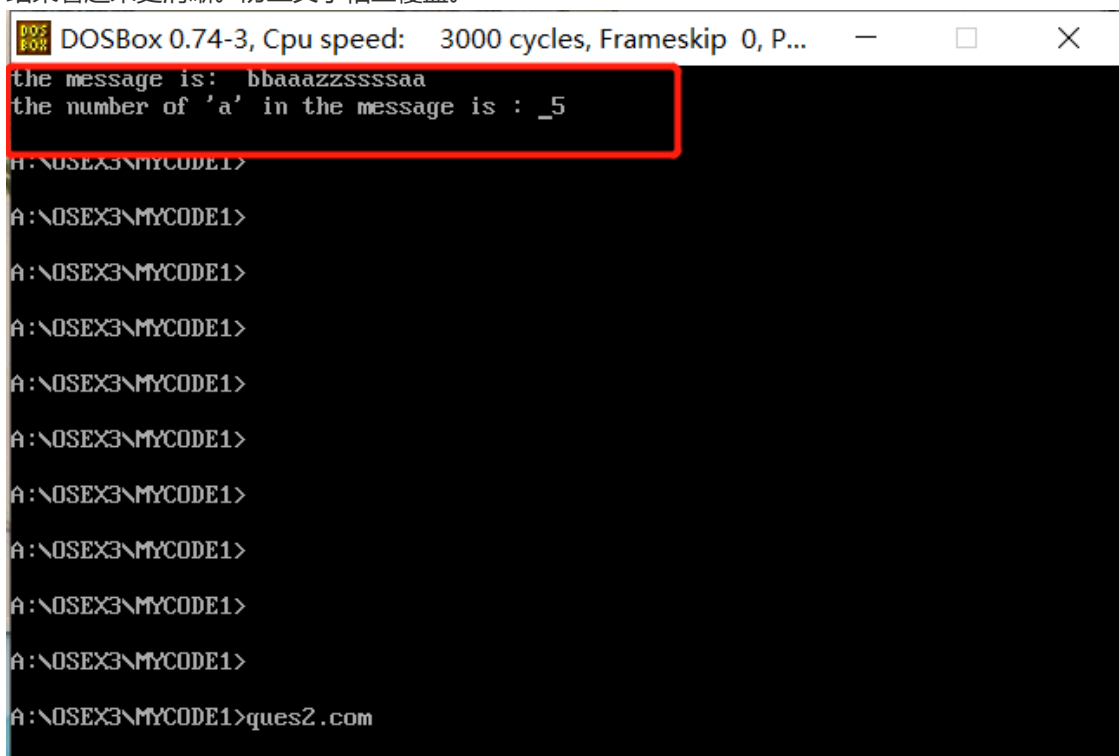
A:\OSEX3\MYCODE1>tasm ques2.asm ques2.obj
Turbo Assembler Version 4.1 Copyright (c) 1988, 1996 Borland International

Assembling file: ques2.asm
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 466k

A:\OSEX3\MYCODE1>tlink ques2.obj ques2c.obj , ques2.com
Turbo Link Version 2.0 Copyright (c) 1987, 1988 Borland International
Warning: no stack

A:\OSEX3\MYCODE1>_
```

5. 运行程序。因为程序运行结果会在界面最上面显示，所以我输入了几个空行把上面的字符清掉，使结果看起来更清晰。防止文字相互覆盖。



```
DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, P...
the message is: bbaaazzssssaa
the number of 'a' in the message is : _5
A:\OSEX3\MYCODE1>
A:\OSEX3\MYCODE1>
A:\OSEX3\MYCODE1>
A:\OSEX3\MYCODE1>
A:\OSEX3\MYCODE1>
A:\OSEX3\MYCODE1>
A:\OSEX3\MYCODE1>
A:\OSEX3\MYCODE1>
A:\OSEX3\MYCODE1>
A:\OSEX3\MYCODE1>ques2.com
```

可以看到运行结果正确。

(三)、监控程序从引导程序分离独立

这一部分真的太南啦啦啦啦啦！

实验步骤

因为要分离内核程序，所以引导扇区目前只需要一个简单的引导，而监控程序要放到一个新的位置。

- 编写引导扇区
- 编写内核C程序

- 编译程序，链接
- 将程序写到软盘中

实现细节

在刚开始的时候使用的是TCC+TASM，但是我不得不说TCC+TASM的方案过于老旧，语法也很复杂，真的很不好用。所以在最后一个部分尝试了很久的TCC+TASM后还是决定换成GCC+NASM。果然NASM的语法上简单了很多。另外在C语言中使用了汇编内联函数，使用的地方不多，觉得这样使用更加方便。整个内核的主体还是在汇编语言中的，但是我觉得这个是不合适的，日后更加复杂的内核设计一定是要在C语言中完成的，觉得内核主体放在C程序中才是更加合适的。下面我们来看一些实现方面的细节，首先来看C程序

内核C程序

我们在C程序中实现了一些基本的输入输出函数，包括输出一个字符，输入一个字符，输出一个字符串，输入一个字符串这些处理指令基本的函数。

getch() 输入一个字符

```
char getch(){
    char ch;
    asm volatile(
        "int 0x16\n"
        : "=a"(ch)
        : "a"(0x1000));
    return ch;
}
```

这里就使用了内联汇编，因为不允许调用库函数，所以我们这里只能用最底层的方法实现。这里就是调用了0x16号中断输入一个字符。另外我们还要说一下内嵌汇编的用法。

带有C/C++表达式的内联汇编格式为：

```
__asm__ __volatile__("InStruction List" : Output : Input : Clobber/Modify);
```

`__asm__` 是GCC关键字asm的宏定义：`#define __asm__ asm`

`__asm__` 或asm用来声明一个内联汇编表达式，任何内联汇编表达式都是以它开头，必不可少。

插入C语言的一个汇编语言代码片段可以分成4部分，以“:”号加以分隔，其一般形式为：

指令部：输出部：输入部：损坏部

指令部

数字加上前缀%，如%1、%0等，表示需要使用寄存器的样板操作数。可以使用的此类操作数的总数取决于具体CPU中通用寄存器的数量。

输出部

规定对输出变量，即目标操作数如何结合的约束条件。每个这样的条件称为一个“约束条件”。必要时输出部可以有多个约束条件，互相以逗号分隔。每个输出约束以“=”号开头，然后是一个字母表示对操作数类型的说明，然后是关于变量结合的约束。

输入部

输入部约束的格式和输出约束相似，但不带“=”号。

损坏部

有些操作中，除了用于输入操作数和输出操作数的寄存器外，还要将若干个寄存器用于计算或者操作的中间结果。这样，这些寄存器原有的内容就损坏了，所以要在损坏部对操作的副作用加以说明，让gcc采取相应的措施。不过，有的时候就直接把这些说明放在输出部了，那也并无不可。

表示约束条件的字母有很多，主要有：

字母	含义
"r"	表示任何寄存器
"m"、"v"和"o"	表示内存单元
"q"	表示寄存器eax, ebx, ecx, edx之一
"i"和"h"	表示直接操作数
"E"和"F"	表示浮点数
"g"	表示任意
"a","b","c" "d"	分别表示要使用寄存器eax ebx ecx和edx
"S"和"D"	分别表示要使用寄存器esi和edi
"l"	表示常数(0至31)

关于内联汇编还有很多知识点，不过我们这次至于到了这些，剩下的的我们以后遇到了再记录。

putch(char ch) 输出一个字符

```
void putch(char ch){
    static int x = 0, y = 0;
    if(ch!='\r'&&ch!='\n'){
        //输出字符
        asm volatile(
            "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) | ch)
            :);
        y++;
        if(y>=80){
            x++;
            if(x>=25) x = 0;
            y = 0;
        }
        //置光标
        asm volatile(
            "int 0x10\n"
            :
            : "a"(0x0200), "b"(0), "d"((x << 8) | y));
        return;
    }else{
```

```

do{
    //输出一行的空格
    putchar(' ');
}while (y);
}
}

```

这一部分函数在编译的时候一直报错!!! 最后是因为gcc默认的内嵌汇编语法是AT&T, 在编译时要添加 `-masm=intel` 换成 `nasm` 的语法。这里真的很坑, 改了很久才改出来。

gets(char* a) 读入字符串

```

void gets(char* s){
    while(1){
        *s = getch();
        putchar(*s);
        if(*s=='\r' || *s=='\n'){
            break;
        }
        s++;
    }
    *s = '\0';
}

```

这里其实是调用了getch()函数, 程序很简单, 不过多赘述。

puts(char* s)输出字符串

```

void puts(char* s){
    while(*s){
        putchar(*s);
        s++;
    }
}

```

这里的程序也很简单只要调用输出字符操作将每一个字符都输出就可以了。

因为我们主要是使用C程序进行用户和程序之间的交互, 所以主要设计的部分字符串的处理部分, 而在汇编程序中进行的是根据传入的指令进行程序的调度。因为有了C语言程序中定义的函数, 所以读入字符串指令可以直接进行, 方便了很多。

编译程序

程序书写完成之后就要进行编译, 之前也提到了在编译的过程出现了很多莫名其妙的错误, 包括但不限于如下所示

```

gardenia@gardenia-virtual-machine:~/OS/EX3$ gcc -march=i386 -m32 -mpreferred-stack-boundary=2 -ffreestanding -c kernel.c -o kernel.o
kernel.c: Assembler messages:
kernel.c:9: 错误: operand size mismatch for `int'
kernel.c:22: 错误: too many memory references for `mov'
kernel.c:23: 错误: invalid char '[' beginning operand 1 `[es:bx]'
kernel.c:36: 错误: operand size mismatch for `int'

```

```

gardenia@gardenia-virtual-machine:~/OS/EX3$ gcc -march=i386 -m32 -mpreferred-stack-boundary=2 -ffreestanding -c kernel.c -o kernel.o
kernel.c: Assembler messages:
kernel.c:22: 错误: too many memory references for `mov'
kernel.c:23: 错误: junk `:[bx]` after expression
kernel.c:23: 错误: too many memory references for `mov'

```

因为中间也该过很多次指令，但是都没有成功，最后看资料才知道因为gcc默认的内嵌汇编语法是AT&T，在编译时要添加 `-masm=intel` 换成 `nasm` 的语法。所以最后这样执行

```
gardenia@gardenia-virtual-machine:~/OS/EX3$ gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -fno-PIE -masm=intel -c kernel.c -o kernel.o
```

编译正确~

之后的任务是编译汇编语言的内核程序。编译汇编语言遇到的问题相对较少，值得注意的一点是，如果用编译NASM的话，在程序的最开始是不需要加上org指令的，所以如果在程序的最开始有org指令，程序会报错，如果删掉程序就可以正确执行

```
gardenia@gardenia-virtual-machine:~/OS/EX3$ nasm -f elf32 moni.asm -o moni.o
moni.asm:9: error: parser: instruction expected
gardenia@gardenia-virtual-machine:~/OS/EX3$ nasm -f elf32 moni.asm -o moni.o
```

如上所示，第一次编译的时候因为没有删掉org指令程序报错，第二次编译的时候删掉之林那个编译成功。

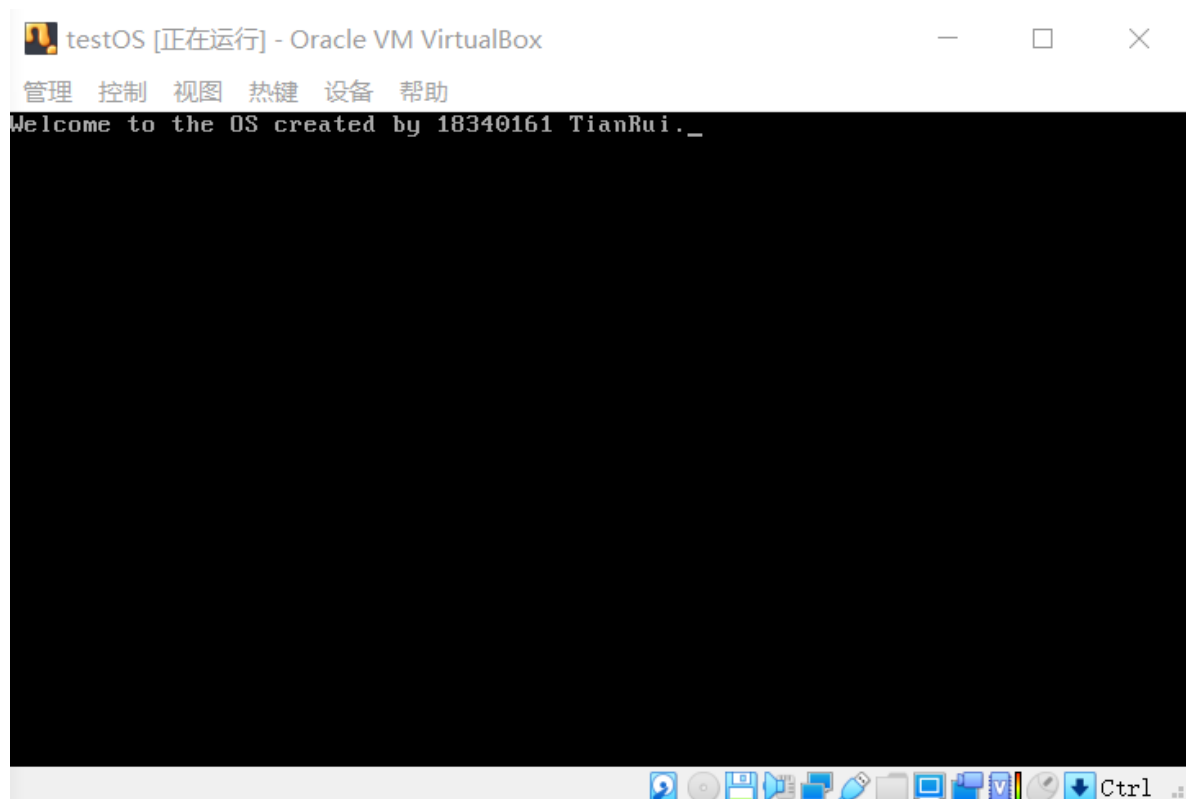
在编译成功之后，输入指令将两个程序链接在一起

```
gardenia@gardenia-virtual-machine:~/OS/EX3$ ld -m elf_i386 -N moni.o kernel.o -Ttext 0x100 --oformat binary -o kernel.bin
```

在程序中我们也可以看出，指令的加载位置是在命令行中指出的，而不是在代码中。

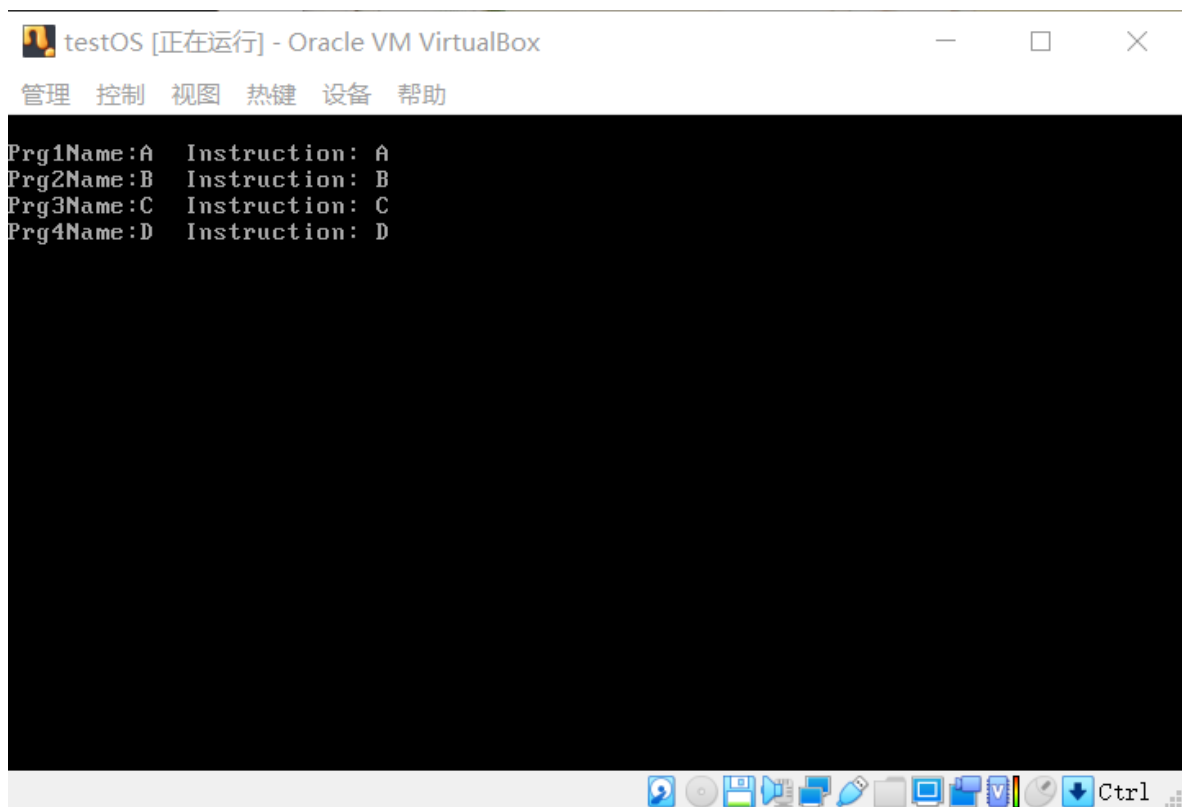
运行结果

完成之后，我们只要想实验二的操作一样用Winhex将程序写进软盘即可。将软盘挂在到虚拟机上运行可以得到运行图片

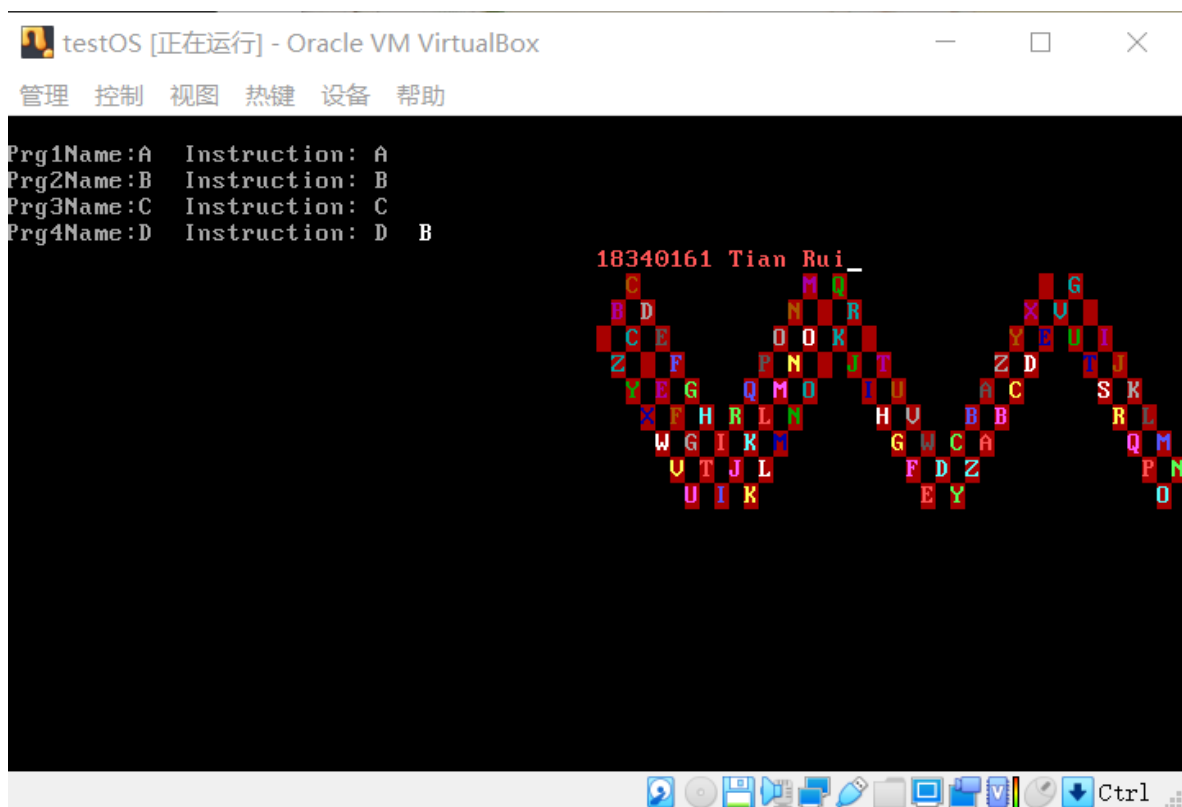


首先是引导扇区，设计成的效果是输出简单的提示字符，在用户按下任意键后跳转到内核程序并清屏。

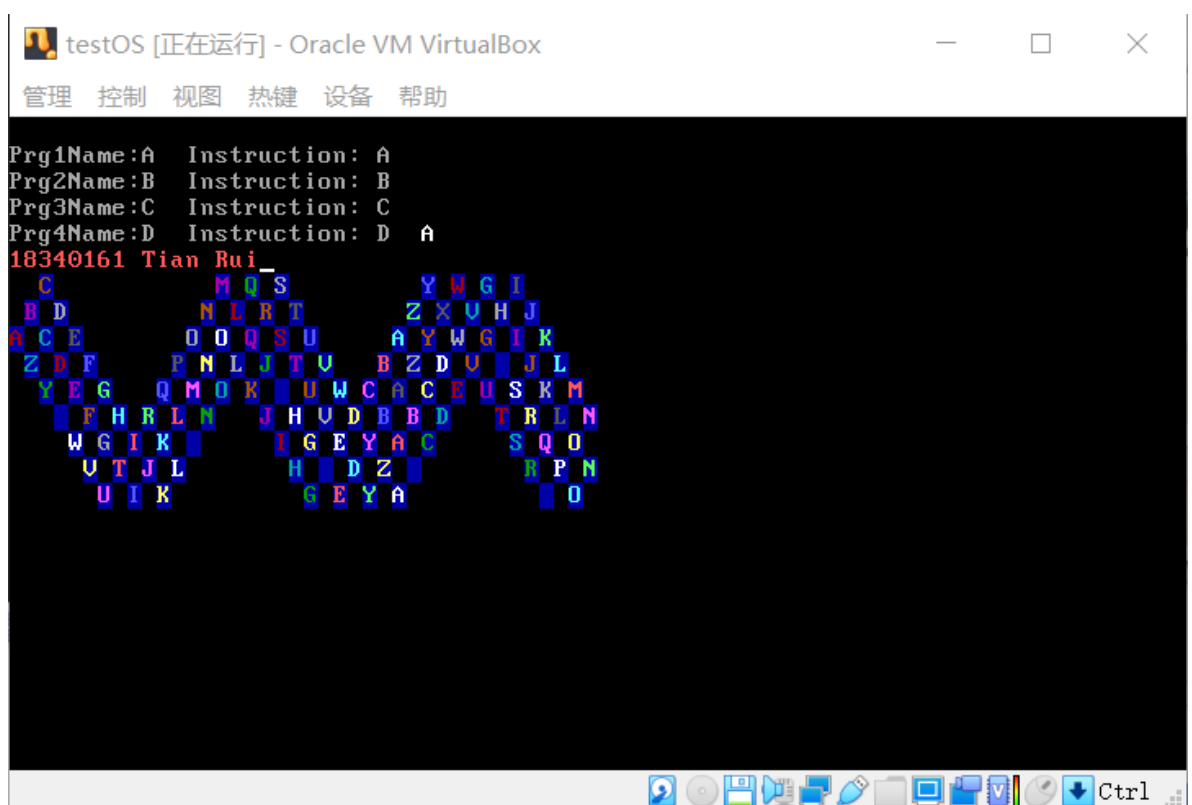
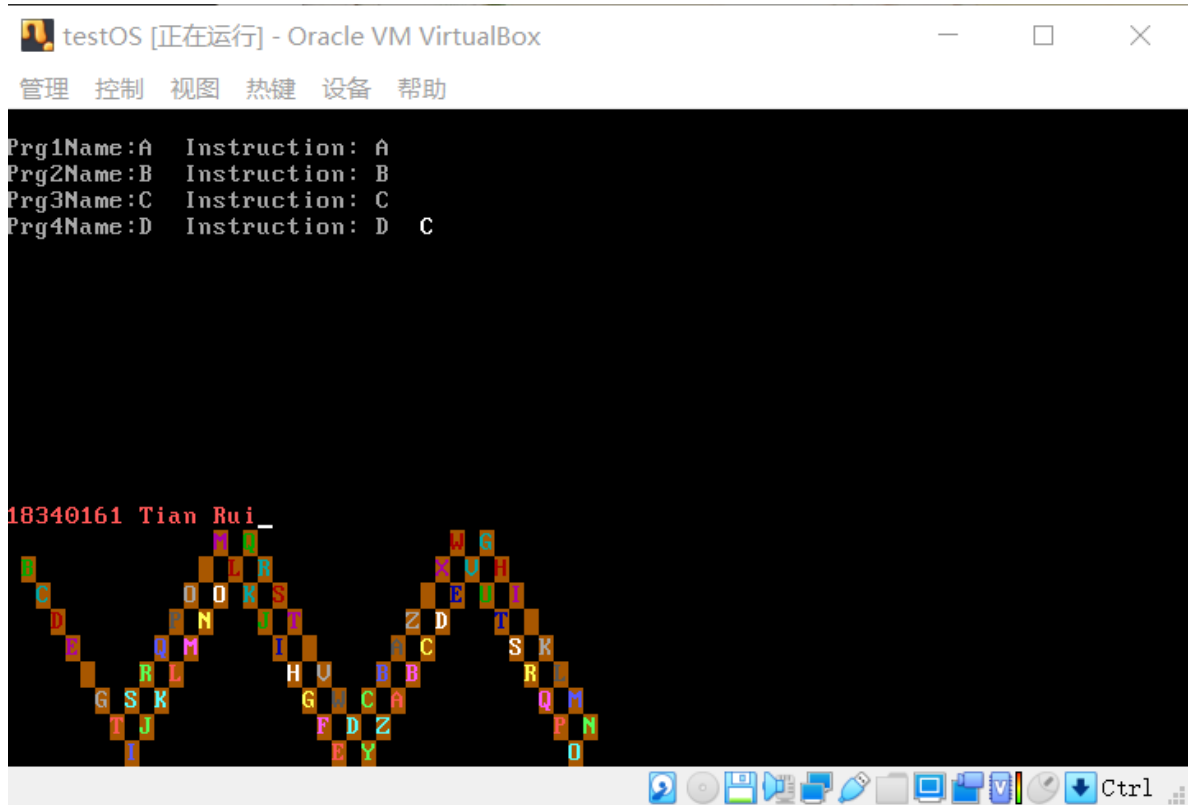
内核程序的效果如下所示，可以输出简单的程序信息。通过输入指令也可以实现程序的调用。

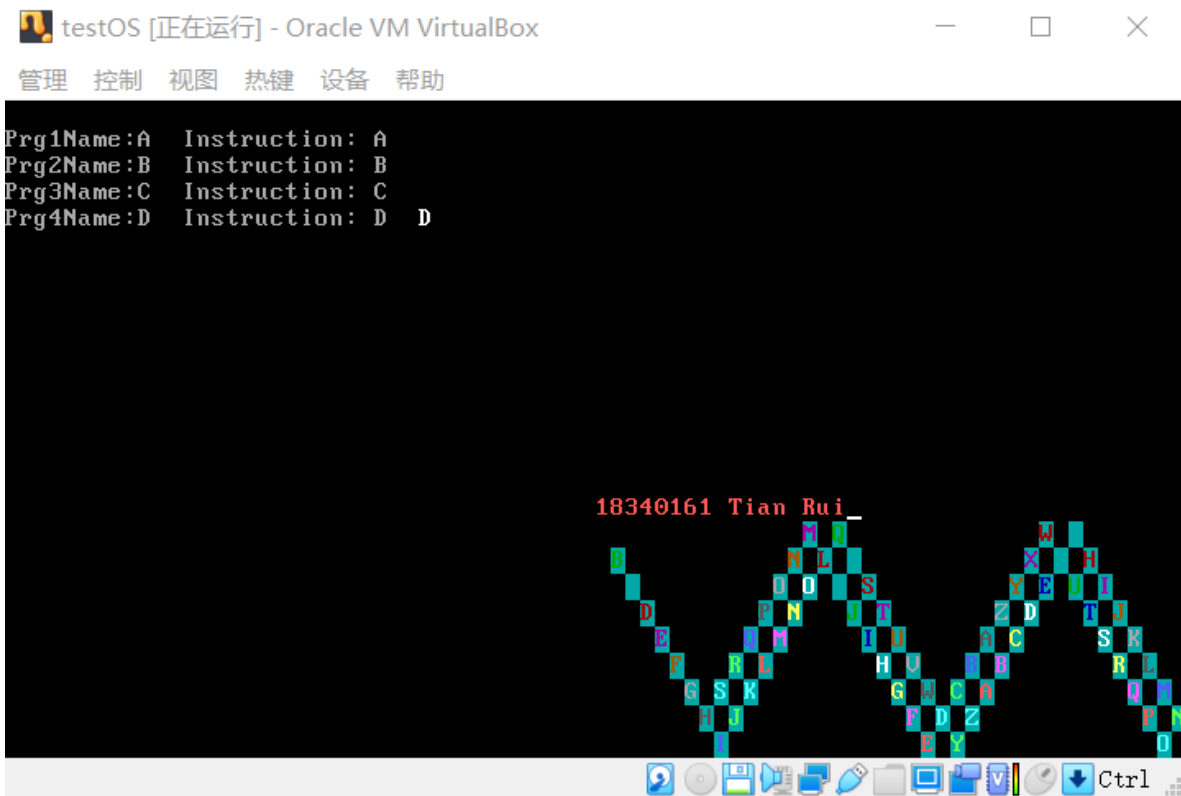


输入指令后可以看到运行效果



一个程序运行结束会再次回到内核程序，再次输入指令即可重复运行。





六、实验总结

(一)、遇到的问题

1. 在分析C语言编译得到的汇编语言时遇到了一个问题。我们在C语言那种定义的是一个大小为2的int型的数组并初始化为0，那么在汇编语言中就应该是 `dw 2 dup (0)`，但是在程序中实际上是却是 `db 2 dup (0)`，最后也没能解决这一问题，希望老师能够帮忙解答。
2. 在实现一个C语言和汇编程序混编的简单案例的时候遇到了很多困难。其中一些问题最终也没有解决。首先我不清楚DGROUPE究竟什么时候要添加，经过多次尝试，如果不添加就不能正确输出，但是却并不理解这里的含义。另外在我debug的时候，C语言编译出来的程序一直存在逻辑错误，但是这段C语言代码用普通的IDE是可以使用的，运行结果也都正确，所以我不清楚这是因为什么。
3. 在编译C语言文件的时候，总是会报一些莫名其妙的错，怎么改都改不好。后来发现是因为编译标准的问题，在编译指令的时候要指明语法标准才可以。
4. 开始刚刚写的时候，不懂怎么用C语言实现输入输出功能，后来经同学指点，是要利用输出字符的方法实现输出字符串，事实上也可以在利用内联汇编在程序中调用中断，但是感觉两种差别其实并不是很大。

(二)、不足

这次试验真的有太多不足了，有很多地方都没有弄清楚，做出来的程序也不是很完美，下面就一点一点的列出自己还可以改进的地方

1. 开始使用TCC+TASM的方法实现，但是TCC的语法太过晦涩，网上的资料也很少，导致最后不得不使用GCC+NASM的方法实现。最后没能使用TCC实现是很遗憾的一件事情
2. 指令设计十分简单。这次的指令仍然只是简短的字符，实际上有了C语言程序是可以实现更加复杂的指令的，但是由于时间有限没能更加深入的实现这一功能。
3. 不知道这一点算不算缺点，但是没有使用纯粹的C语言和汇编语言的链接，而是使用了内联汇编这种讨巧的方法。
4. 没有把子程序用C语言表示。没有将被调用的子程序也用C语言表示出来再链接到一起。

总之这次试验有太多太多的不足了，自己要学习的只是还很多，以后还要继续努力呢！

七、参考文献

- [1]wu-kan.用汇编与C语言开发独立内核[EB/OL].https://wu-kan.cn/_posts/2019-03-28-%E7%94%A8%E6%B1%87%E7%BC%96%E4%B8%8E%E8%AF%AD%E8%A8%80%E5%BC%80%E5%8F%91%E7%8B%AC%E7%AB%8B%E5%86%85%E6%A0%B8/,2019-3-28.
- [2]robbie1314.asm volatile 之 C语言嵌入式汇编[EB/OL].https://blog.csdn.net/robbie1314/article/details/6329329?utm_medium=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.nonecase&depth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.nonecase,2011-04-17.
- [3]老鼠爱旅游.asm volatile内嵌汇编用法[EB/OL].https://blog.csdn.net/whut_gyx/article/details/39078339,2014-09-05.
- [4]nemonic27.【Linux内核分析】【第四周作业】使用嵌入式汇编进行系统调用[EB/OL].<http://blog.chinaunix.net/uid-13980029-id-4924025.html>,2015-03-29.
- [5]程序员cxuan.程序员需要了解的硬核知识之汇编语言(一)[EB/OL].<https://www.cnblogs.com/cxuanBlog/p/11941526.html>,2019-11-27.