实验二 加载用户程序的监控程序 实验报告

数据科学与计算机学院　2017 级计算机科学与技术　1734146　王程钥

1　实验题目

加载用户程序的监控程序

2　实验目的

掌握监控程序加载用户程序的方法
掌握 BIOS 编程的方法
掌握 DOS 中断命令的使用
熟悉 x86 的语法

3　实验要求

3.1 设计有输出的用户可执行程序

设计四个有输出的用户可执行程序，分别在屏幕 1/4 区域动态输出字符，如将用字符‘A’从屏幕左边某行位置 45 度角下斜射出，保持一个可观察的适当速度直线运动，碰到屏幕相应 1/4 区域的边后产生反射，改变方向运动，如此类推，不断运动；在此基础上，增加你的个性扩展，如同时控制两个运动的轨迹，或炫酷动态变色，个性画面，如此等等，自由不限。还要在屏幕某个区域特别的方式显示你的学号姓名等个人信息。

3.2 设计加载用户程序的监控程序

修改参考原型代码，允许键盘输入，用于指定运行这四个有输出的用户可执行程序之一，要确保系统执行代码不超过 512 字节，以便放在引导扇区。

自行组织映像盘的空间存放四个用户可执行程序。

4　实验方案

4.1 实验环境

编程环境：GCC+NASM
16 进制编辑器：WinHex
虚拟机：WMware Workstation
虚拟机环境

- 操作系统　MS-DOS
- 内存　1MB
- 硬盘　102MB
- 处理器　1 核心
- 虚拟化引擎首选模式　Intel VT-x/EPT 或 AMD-V/RVI

4.2 监控程序和用户程序的交互

将监控程序和用户程序分别在 NASM 环境下编译成 com 文件，将两者的二进制编码合并

在一起，制作成软盘镜像文件，并使用 VMware 虚拟机加载对应的软盘镜像文件。

监控程序需要根据键盘输入调用对应的用户程序并执行相关程序。

用户程序的过程中，需要根据键盘输入暂停，继续运行，返回监控程序。

4.3 汇编代码重构

老师给的 stone.asm 程序实现比较复杂，编译产生的 com 文件大小已经接近 512K 的上界，不太方便后序操作。为了方便操作，需要对 stone.asm 程序进行重构，简化程序实现过程，从而减小 com 文件大小，方便加入其它语句，实现和监控程序的交互。

5 实验过程

5.1 使用监控程序加载用户程序

首先我使用了老师给的 myos1.asm 的汇编代码，尝试调用实验 1 写的反弹程序。我将监控程序中的 OffSetOfUserPrg1 域定义为 0a100h，并在用户程序中加上 org 0a100h。我直接将两者编译出来的 com 文件合并，放入虚拟机中运行。

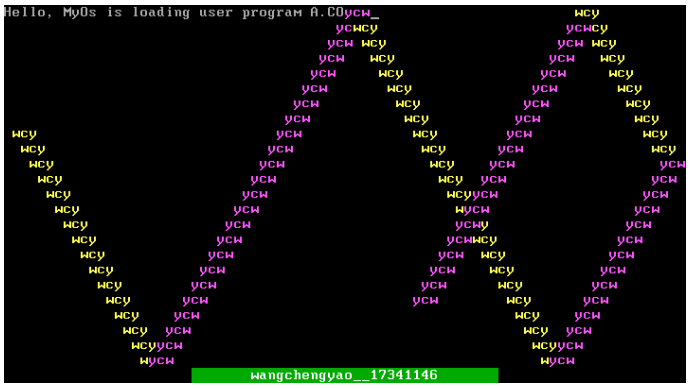

图 1

如图，用户程序成功被监控程序引导

5.2 使用监控程序加载多个用户程序

我继续尝试往扇区中加入其他程序。我在扇区中又加入了老师给的 stone.asm 程序对应的 com 文件，并将监控程序中的扇区改成 3（第一个扇区是引导扇区，从 2 开始计数）。我将其制作成软盘镜像文件并运行虚拟机尝试运行。但很遗憾虚拟机中却不能如愿地运行对应的程序（如图 2）。

图 2

我检查了一下代码，并没有发现错误。我将监控程序中的扇区改成 2，是可以运行的。

我查看了一下 test.flp 文件的 16 进制编码，发现监控程序对应的 16 进制编码中有很多 0,并在最后有 55,aa（如图 3）。

我观察了一下，这段代码总共长 512KB，刚好是一个扇区的大小。后面两段代码是直接贴在一起的，中间没有间隔（如图 4）。

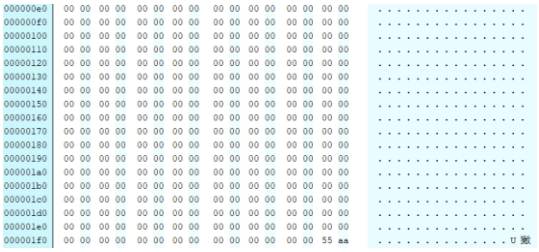

图 3

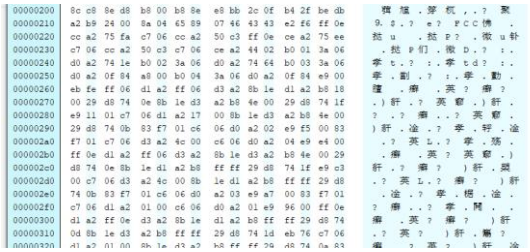

图 4

我继续观察了一下 myos1.asm 的源代码，发现了这两句话

```
times 510-($-$$) db 0
db 0x55,0xaa
```

这两句话的意思是将二进制文件补满到 512KB，并在最后加上 0x55,0xaa。我上网查了一下，0x55,0xaa 这两句话的意思是一个扇区的终止符号。我的两个扇区没有加上终止符号，BIOS 也就不会认为我的这两段代码属于两个扇区。

于是我将两段用户程序的最后分别加上了上述的两句话，再装入软盘运行。结果，虚拟机却还是没能成功运行 stone.asm 对应的反弹程序（运行结果与图 2 相同）。

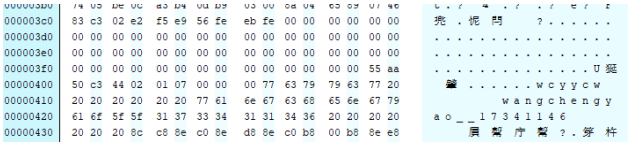

图 5

我观察了一下 flp 文件，发现在 0x55,0xaa 后出现了数据段的定义（如图 5）。数据段是放置在正文段之后的，两者并没有放在同一个扇区里。我将 section.data 删掉，直接将两者放在一起，再编译运行。

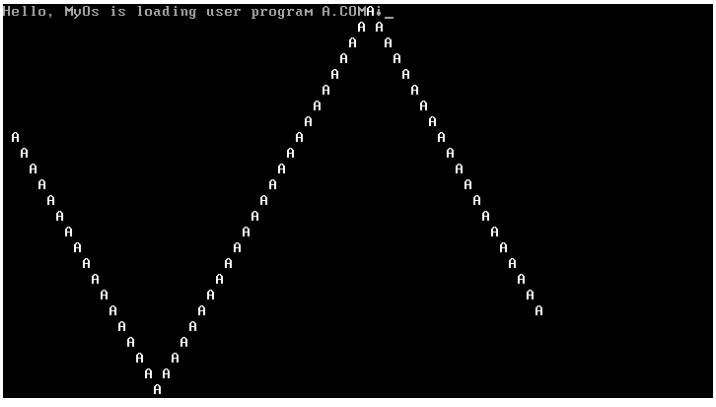

图 6

如图 6，监控程序成功引导了 stone.asm 对应的反弹程序。

因此，我修改了一下合并 com 文件的 cpp 代码。我在每段代码最后都机上上述两句话，并直接将 com 文件对应的二进制编码拼接起来。相关代码如下。

```
#include<bits/stdc++.h>
#define N 1440*1024
```

```
using namespace std;
char s[N*2],ch;int tot=0;
int main()
{
  FILE *fp,*des;
  fp=fopen("test.com","rb");
  while(fread(&ch,sizeof(char),1,fp))s[tot++]=ch;
  fclose(fp);
  fp=fopen("a.com","rb");
  while(fread(&ch,sizeof(char),1,fp))s[tot++]=ch;
  fclose(fp);
  fp=fopen("b.com","rb");
  while(fread(&ch,sizeof(char),1,fp))s[tot++]=ch;
  fclose(fp);
  fp=fopen("c.com","rb");
  while(fread(&ch,sizeof(char),1,fp))s[tot++]=ch;
  fclose(fp);
  fp=fopen("d.com","rb");
  while(fread(&ch,sizeof(char),1,fp))s[tot++]=ch;
  fclose(fp);
  des=fopen("test.flp","wb");
  fwrite(s,sizeof(char),N,des);
  fclose(des);
  return 0;
}
```

5.3 汇编代码的重构

我本来打算将实验 1 的代码进行简单修改后用于实验 2 的四个用户程序。但我发现实验 1 的反弹程序编译后的大小已经接近 512KB，不方便再进行额外的修改。于是我决定将反弹程序的代码进行重构。

实验 1 的代码是在老师给的 stone.asm 上进行修改的。老师的代码是将反弹程序分为右上、右下、左上、左下这四个部分。实际上，这四个部分本质上是相同的。不同的地方仅仅在于每一步 x,y 坐标变化的不同（见表 1）。

	左上	左下	右上	右下
x	-1	+1	-1	+1
y	-1	-1	+1	+1

表 1

所以我决定使用变量 xx,yy 来存储上述数据，并使用 x,y 在存储当前坐标。对于每一步操作的坐标改变，我们采用如下的代码。

```
mov bx,word[xx]
add word[x],bx
mov bx,word[yy]
add word[y],bx
```

对于边界判定，我每移动一次就判断一下当前坐标是否会超出上下左右这四个边界。具体判定代码如下（以左边界为例）。

```
check_LR:
    mov bx,word[y]
    mov ax,L
    sub ax,bx
      jz modify_L
(check_R 略)
modify_L:
    mov si,1
    mov word[y],L+2
    mov word[yy],1
    jmp check_UD
```

其中 si 变量为标记变量，标记当前步骤是否改变方向（该变量用于判断输出颜色是否需要改变）。

以下为重构后的代码主要框架。

```
start: 开始
work: 改变坐标

check: 碰撞检测
  check_LR: 判断是否碰上左右边界
    if 碰撞 L: modify_L
    if 碰撞 R: modify_R
  check_UD: 判断是否碰上上下边界
    if 碰撞 U: modify_U
    if 碰撞 D: modify_D
  show

modify: 修改
  modify_L: 碰左边界的修改
    check_UD
  modify_R: 碰右边界的修改
    check_UD
  modify_U: 碰上边界的修改
    show
  modify_D: 碰下界的修改
    show

Show: 显示字符
End: 结束
```

通过简单的重构，编译后的 com 文件大小得到了大幅度的下降，可以开始快乐地魔改代码了！

5.4 监控程序(myos.asm)的编写

根据实验要求，我需要设计四个有输出的用户程序程序，并分别在屏幕的 1/4 区域运行。我需要设计一个监控程序，通过输入控制用户程序的加载。

监控程序放在引导扇区，需要完成输出提示字符串，调用扇区的功能。

5.4.1 输出引导字符串

引导字符串有如下三个，第一个为欢迎语句，第二个为输入提示语句，第三为姓名学号的显示语句。

```
Message          db 'Welcome to wcy1122"s OS!'
MessageLength    equ ($-Message)
Hint             db 'Input the guide program you want to run:'
HintLength       equ ($-Hint)
Info             db 'Wang Chengyao  17341146'
InfoLength       equ ($-Info)
```

输出时调用 BIOS 的 10h 号中断，功能号为 13h。行号、列号和输出字体颜色由 5.5 的界面设计决定。以下为输出引导字符串的相关代码。

```
print_str:
    mov bp,Message          ; BP=当前串的偏移地址
    mov ax,ds               ; ES:BP=串地址
    mov es,ax               ; 置 ES=DS
    mov cx,MessageLength    ; CX=串长
    mov ax,1301h            ; AH = 13h(功能号)、AL=01h(光标置于串尾)
    mov bx,0047h            ; 页号为 0(BH=0) 红底白字(BL=47h)
    mov dh,11               ; 行号
    mov dl,28               ; 列号
    int 10h                 ; BIOS 的 10h 功能:显示一行字符
    mov bp,Info             ; BP=当前串的偏移地址
    mov cx,InfoLength       ; CX=串长
    mov dh,13               ; 行号
    mov dl,28               ; 列号
    int 10h                 ; BIOS 的 10h 功能:显示一行字符
    mov bp,Hint             ; BP=当前串的偏移地址
    mov cx,HintLength       ; CX=串长
    mov dh,12               ; 行号
    mov dl,20               ; 列号
    int 10h                 ; BIOS 的 10h 功能：显示一行字符
```

5.4.2 读取并调用扇区

监控程序需要根据输入的字符确定调用的用户程序的扇区编号。若输入’1’则调用第 2 个扇区，若输入’2’则调用第 3 个扇区，以此类推。

字符输入调用 BIOS 的 16h 号中断，功能号为 0h。该中断可以实现有阻塞的键盘输入。输入后将输入的数据放到 x 变量中。相关代码如下。

```
input:
    mov ah,0                ; AH = 0h(功能号)
    int 16h                 ; BIOS 的 16h 功能：读入下一个按键
```

```
    cmp al,'1'
      jl input
    cmp al,'4'
      jg input              ; 若读入字符不是 1234 则不操作
    sub al,'0'              ; 转换输入的字符
    mov byte[x],al          ; 将输入的字符存入 x 变量
```

调用扇区使用 13h 号中断，功能号为 02h。根据读入的扇区编号 x，加上 1(不调用主扇区)，得到调用的扇区编号并存入 cl 寄存器。相关代码如下。

```
LoadnEx:
    ;读软盘或硬盘上的若干物理扇区到内存的 ES:BX 处
    mov ax,cs                   ;段地址,存放数据的内存基地址
    mov es,ax                   ;设置段地址(不能直接 mov es,段地址)
    mov bx,OffSetOfUserPrg1     ;偏移地址,存放数据的内存偏移地址
    mov ah,2                    ;功能号
    mov al,1                    ;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 byte cl,[x]
    add cl,1                    ;调用扇区号,起始编号为 1
    int 13H                     ;调用读磁盘 BIOS 的 13h 功能
    ; 用户程序 a.com 已加载到指定内存区域中
    jmp OffSetOfUserPrg1
```

5.5 用户程序的修改

被监控程序调用的用户程序需要进行一些修改。包括提示字符串的输出，程序的暂停与继续和返回监控程序的功能实现。提示字符串输出部分与监控程序相关内容的实现方法相同，不再赘述。

5.5.1 用户程序的暂停和继续

用户程序的暂停需要由键盘输入来控制，键盘输入任意键则程序暂停。这个功能需要使用无阻塞的键盘输入。我在网上进行了查找，发现 16h 号中断的 01h 号功能可以实现这个操作[1]。若键盘没有输入则 al=0，若有输入则 al=1。

暂停后，用户程序可以选择继续运行或返回监控程序。这由输入的另一个字符控制，若为’e’则退出，若为’r’则继续。以下为相关代码。

```
loop1:
    ;时间延迟代码,略
    mov ah,1                    ; 功能号=1
    int 16h
    jz work                     ; 键盘无输入,不中断
    jmp check                   ; 键盘无输入,不中断
```

```
check:
    mov ah,0                    ; 等待缓冲区
    int 16h
    cmp al,'e'
      jz end                    ; 输入’e’,退出
    cmp al,'r'
      jz work                   ; 输入’r’,继续运行
    jmp check                   ; 输入非法,不执行操作
```

5.5.2 返回监控程序

一开始我参考了课件中代码，在程序结束点放置了以下代码。

```
mov ax,4c00h
int 21h
```

但很遗憾，通过在虚拟机上的运行，我发现运行这段代码后虚拟机会直接死机，而不会返回监控程序。这条指令的功能是返回 DOS 系统的，但我们在调用用户程序的时候是直接 jmp 到内存入口的。也就是说，调用过程并没有留下返回地址等信息。因此词汇调用 int 21h，会导致死机。

我在网上搜索了相关的资料，试图寻找相关的 BIOS 指令。但很遗憾，我最终也没有找到什么太好的办法。没有办法，我选择了一个可以算是作弊的办法，即在程序结束点后加上 jmp 7c00h，即回到监控程序对应内存入口。

这样写虽然可以完成本次实验的要求，但对于一个操作系统而言其实是很有问题的。首先是用户程序可移植性的降低。用户程序挂载在不同的操作系统上时需要根据对应操作系统的内入口移修改代码，需要和操作系统的设计者进行商量。其次是安全性的问题，操作系统的内存入口一旦被公开，安全性就会受到很大的威胁。黑客能够随意地进入操作系统的内存入口修改内核程序。

5.6 界面设计与个性化拓展

本次实验的总体设计方案如下。

监控程序和用户程序的界面布局相同。界面的中间为信息显示屏，用于相关提示信息的显示和操作信息的输入。界面主要分为四个部分，分别为四个用户程序的显示空间。用户程序为四个反弹程序，每个用户程序都有一个字母在对应的显示空间内反弹。每个用户程序的显示空间分配见表 2。

在监控程序界面，用户可以输入想要运行的扇区编号(1~4)，监控程序会运行相关扇区。

在用户程序界面，用户可以输入任意键使用户程序暂停。暂停后，用户可以输入’r’键恢复运行，或输入’e’键返回监控程序。用户程序每次重新调用会清屏。

用户程序	显示空间（[行范围]x[列范围]）
stone1	[0,10]x[0,38]
stone2	[0,10]x[40,79]
stone3	[13,25]x[0,38]
stone4	[13,25]x[40,79]

表 2

5.6.1 监控程序界面

监控程序界面的设计如下。首先有一个绿色的十字，分割四个用户程序的显示空间。其次在界面的中间有一个红色的显示屏，用于显示姓名学号和相关提示信息。

界面的实现，使用 BIOS 中 10h 号中断的 06h 号功能。实现原理和界面清屏相同，等价于在对应的区域用相关颜色(如绿色，红色)清屏。相关代码将在 5.6.2 中提供并详细说明。字符串的实现已经在 5.4.1 中说明，不再赘述。监控程序界面的效果图可见图 7。

图 7

5.6.2 用户程序界面

用户程序界面的设计大体与监控程序相同，除了输出的提示字符串有小的不同。在同一个区域修改提示字符串，需要先将相关区域清屏，后调用 10h 中断输出字符串。用户程序界面效果图可见图 8，图 9。

四个用户程序都采用了反弹程序，每次碰到边界后字符会变色。具体不同可见表 3。

图 8（运行中的用户程序）

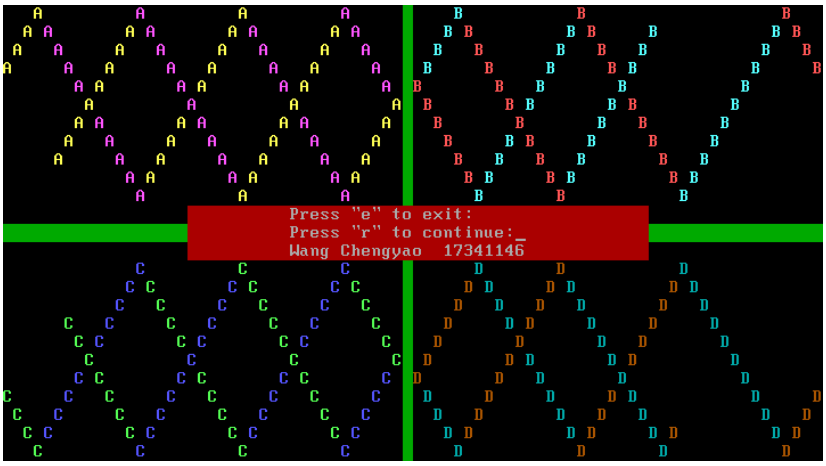

图 9（暂停时的用户程序）

	显示字符	显示颜色	初始运动方向	初始位置
Stone1	A	黄/紫	右上	(3,0)
Stone2	B	红/青	左上	(3,79)
Stone3	C	绿/蓝	左下	(21,0)
Stone4	D	棕/蓝	右下	(21,79)

表 3

5.6.3 界面清屏

每次重新运行相关程序时需要对其显示界面清屏，需要使用 BIOS 中断实现。我上网找到了相关的 BIOS 中断。调用 BIOS 的 10h 号中断的 06h 号功能，即可在相关区域完成清屏[2]。清屏本质上是用空格填满整个区域。界面设计部分相关功能的实现与此相似，只需修改清屏区域和清屏颜色即可。相关代码如下。

```
clear:
    mov ax,cs
    mov ds,ax          ; 数据段
    mov ah,06h         ; 功能号 06h
    mov al,0           ; 清屏字符，0 为空格
    mov ch,U+1
    mov cl,L+1
    mov dh,D-1
    mov dl,R-1         ; 清屏区域[L+1,R-1]x[U+1,D-1]
    mov bh,07h         ; 07h,黑底白字
    int 10h           ; 调用 10h 号中断
```

5.7 运行结果

图 10~图 17 为程序的运行结果。

图 10(初始界面)

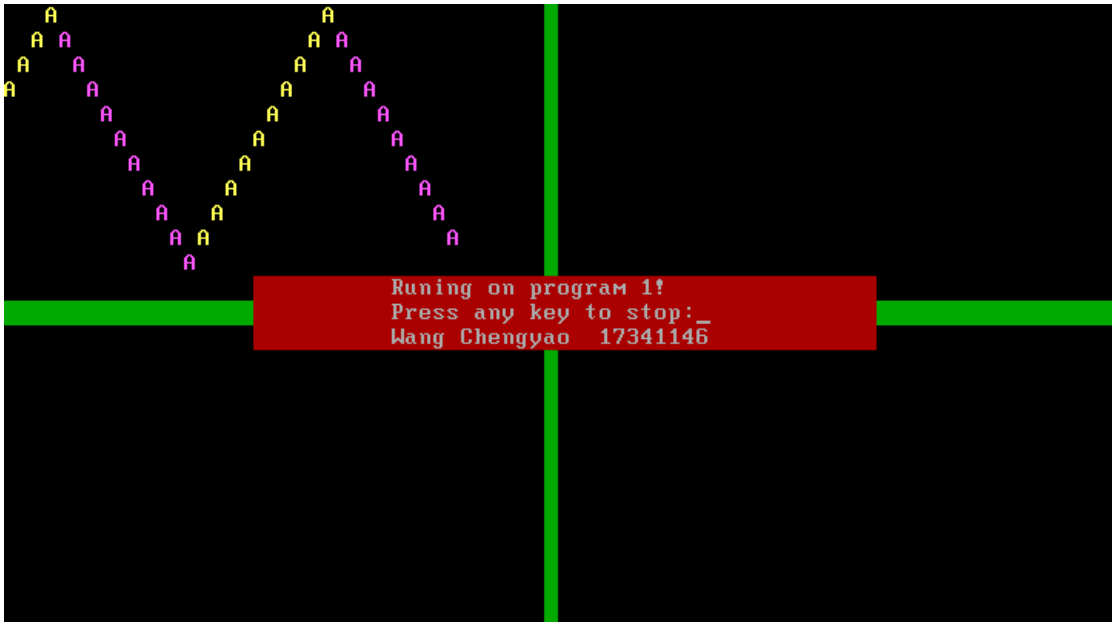

图 11(运行 stone1)

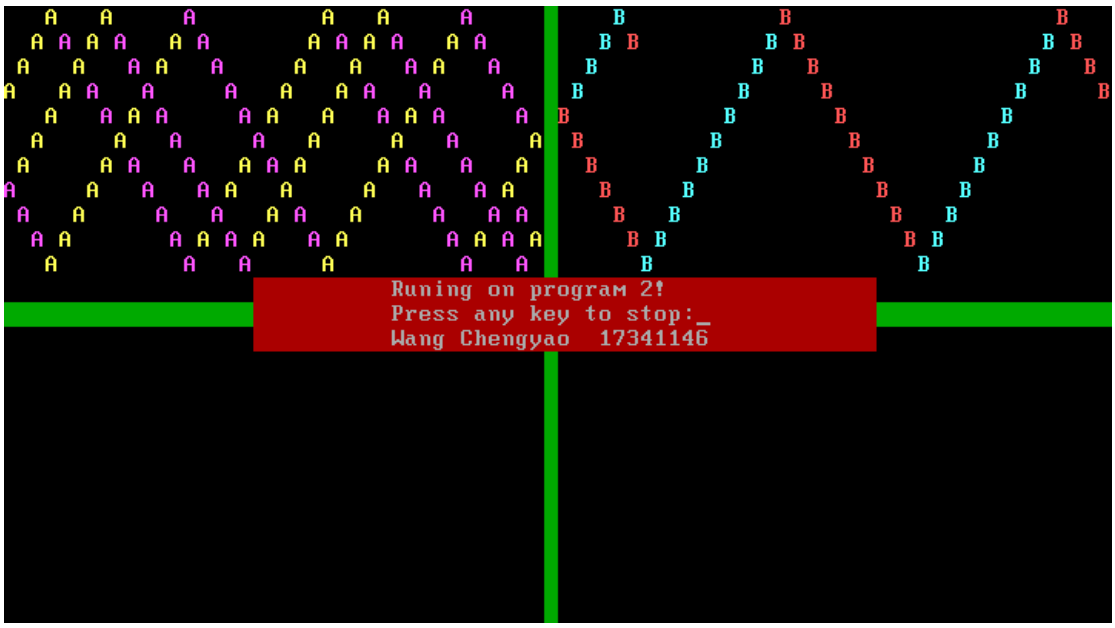

图 12(运行 stone2)

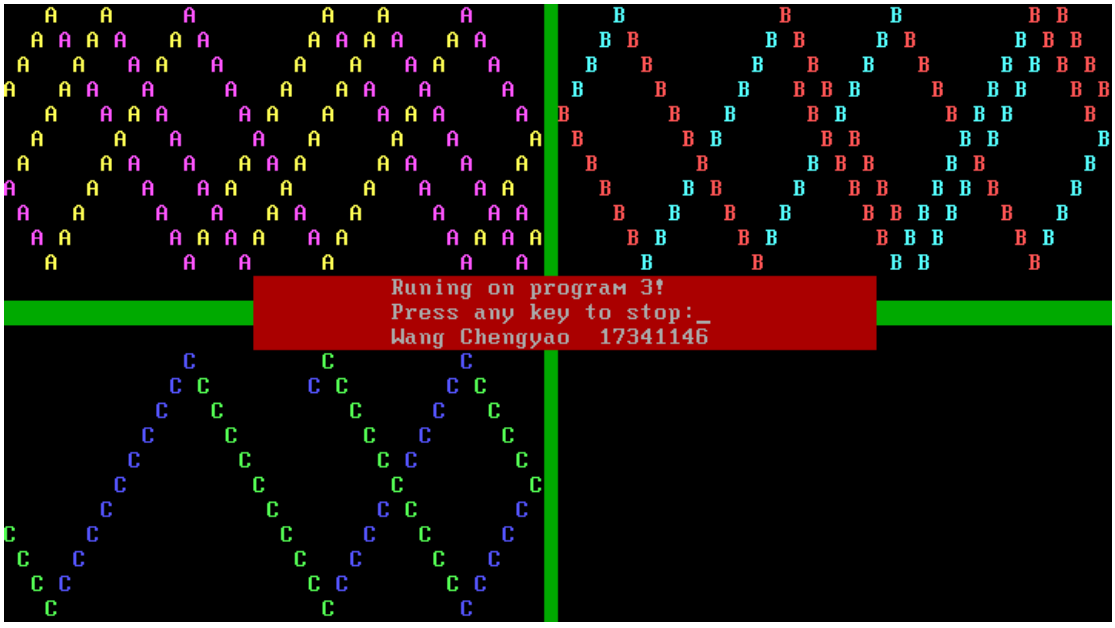

图 13(运行 stone3)

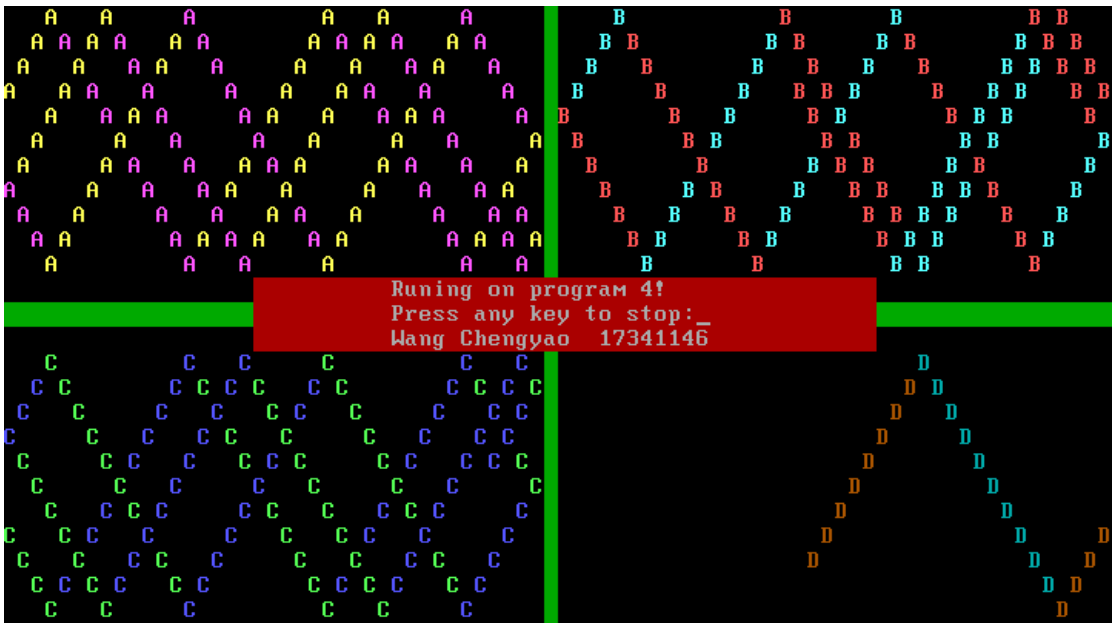

图 14(运行 stone4)

图 15(重新运行 stone1)

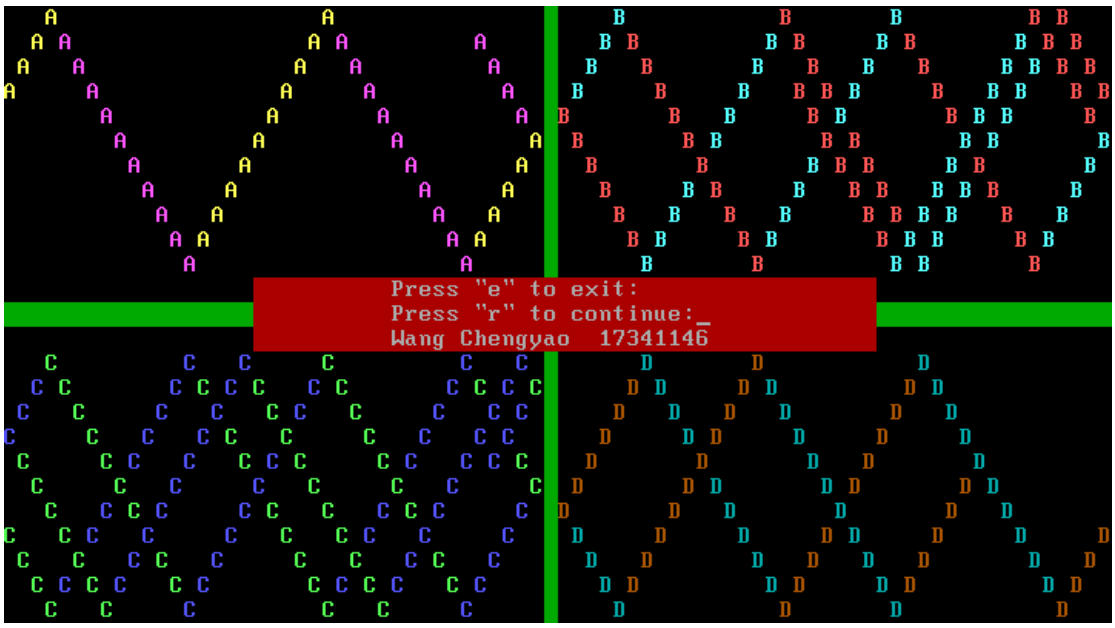

图 16(暂停)

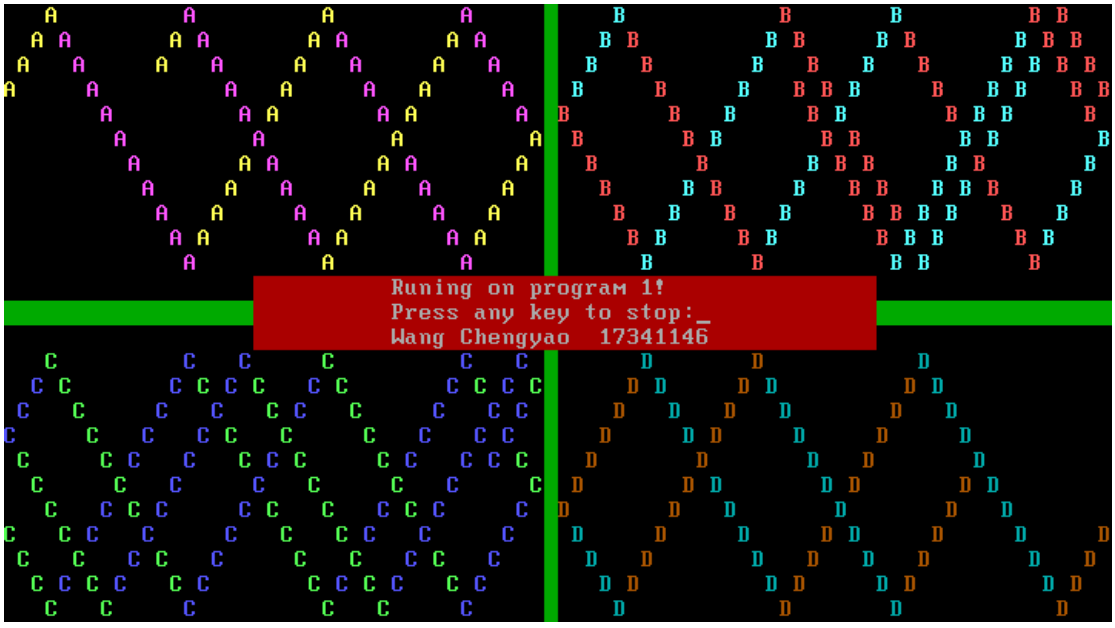

图 17(恢复运行)

6 实验总结

本次实验总体完成得比较成功，基本上实现了实验开始前的规划和设想。本次实验要求设计一个监控程序，实现调用扇区中的用户程序的功能。在实验要求的基础上，我对操作系统的显示页面进行了设计，并在用户程序中添加了暂停，继续和清屏等功能，实现了一些个性化的拓展。

但本次实验还是有几个缺陷。第一个缺陷是，返回操作系统部分很草率地用 `jmp 7c00h` 糊过去了。这确实能够完成实验要求，但作为一个操作系统这么做显然是不合格的。第二个缺陷是，我本来希望用户程序在退出后再次被调用时能够从上次退出的位置开始反弹，即在进程中断后恢复进程。但这需要保存下程序运行的信息，实现起来不是那么容易，也没有找

到 BIOS 的相关中断程序。因此最终我放弃了这个需求。完成这个需求所需要的只是应该会在实现多进程操作系统的时候学到，到那个时候可以考虑再加上这个功能。

完成实验后我和刘瀚之同学进行了交流，获得了一种更有优秀的返回监控程序的方法。在加载引导程序的时候不直接 jmp 过去，而使用 call 语句。Call 语句在跳转的时候会记录下返回地址。这样从用户程序返回监控程序的时候调用 ret 语句就可以完成。这样做显然提高了操作系统的安全性和可移植性。

通过本次实验，我掌握了 BIOS 编程的语句，尝试使用了一些 BIOS 中断语句，对操作系统的中断有了更深入地理解。同时，通过重构 stone.asm 的代码，我更加熟悉了 x86 的语法，熟悉了 NASM 的编译环境。另外，在实验过程中，我发现了实验 1 中存在的一些漏洞和问题，并对其进行了一些改进。最后，我还更加熟悉了从汇编到 16 进制编辑器到虚拟机的代码编辑环境。

本次实验实现了一个可以切换进程的监视器，实现了程序和键盘的交互，进行了界面的设计。这也可以算是实现了一个简单的操作系统了。

参考文献

[1]键盘输入 https://blog.csdn.net/qingkongyeyue/article/details/68490194
[2]清屏 https://blog.csdn.net/mxue_ncpss/article/details/85992088
[3]BIOS 中断
https://blog.csdn.net/piaopiaopiaopiaopiao/article/details/9735633