监控程序实现控制用户程序

学院名称：	数据科学与计算机学院
专业（班级）：	18计科教学3班
学生姓名:	夏一溥
学号:	18340178
时间:	2020 年 5 月 8 日
实验二：	监控程序

实验内容

- [x] 修改实验一，完成4个不同版本程序分别在1/4屏幕区域显示，并在1.44MB软驱映像中存储这些程序
- [x] 重写1.44MB引导程序，利用BIOS调用实现能加载COM格式用户程序的监控程序
- [x] 设计命令，实现交互执行在软驱上的用户程序
- [ ] 在映像盘上设计表格记录盘上用户程序信息

实验过程

实验思路

1.设计四个独立的用户程序:

用户程序可以仿照实验一完成的程序，而为了控制字符分别在各自四分之一区域里运动，只需要规定字符移动的范围边界，当触碰边界时就完成反弹。

用左上角第一个小矩形的用户程序为例:

```
    screen_left equ -1
    screen_top equ -1
    screen_right equ 40
    screen_bottom equ 13
```

通过以上代码可以设置字符反弹时的边界，从而控制字符只在指定区域运动。

2.Boot程序:

由于逻辑开始执行用户程序的起始地址为07C00h，于是这里第一个运行的程序为引导程序，需要引导程序能够将后续需要执行的应用程序1-4预先装填与内存区域。而由于现在实现的操作系统没有后台运行程序这一概念，所以这里我有两种思路:

1. 将不同的用户程序装在在各自的内存区间，在监控程序检测到用户需求是跳转到相应的地址;
2. 将引导程序与监控程序合二为一，每当检测到用户需要加载某一程序，再将这一程序在固定地址加载执行。

这两种解决方案各有利弊，第一种不需要花费加载的时间，并且后续处理方便，缺点也很明显，占用更大的内存空间。第二种则胜在空间利用率高，可以执行需要内存更大的程序，缺点则是当频繁切换运行用户程序时加载耗时高。我选择了第一种方案。

加载用户程序选择使用13号中断，其功能为读软盘或硬盘上的若干物理扇区到内存的ES:BX：

```
%macro read_secotr 6  ;(offset,扇区数,驱动器数,磁头号,柱面号,起始扇区号)
    mov ax,cs
    mov es,ax
    mov bx,%1
    mov ah,2                ；功能号
    mov al,%2
    mov dl,%3
    mov dh,%4
    mov ch,%5
    mov cl,%6
    int 13H ;
%endmacro
```

同时加载各不同的用户程序到各自内存区域就反复调用宏即可。同时需要生成表来记录用户程序信息，就在数据段附加：

```
num_prog dw 0
prog_pos dw 0
```

来记录程序数和程序地址。

3.monitor程序：

监控程序负责接受用户按键来选择跳转到对应的用户程序所在内存区域。

其中，识别用户输入使用16H中断：

```
mov ah, 0x01
    int 16h
    jz ini
    mov ah, 0
    int 16h
    cmp al, 'n'
    je _progn
```

ah为16H所需要的参数，01为非阻塞且检查键盘是否输入，有输入则标志位置零，00则阻塞且读取键盘输入，将读取到的值存放于ax.

为了使用户程序运行后回到监控程序，选择使用call：

```
_prog1:
    call prog1
    jmp ini
```

再返回ini进行下一条命令的读取。

4.将用户程序1~5、监控程序、boot程序分别编译并写入软盘

空软盘制作使用 Linux 下 dd命令完成：

```
d if=/dev/zero of=floppy.img bs=512 count=2880
```

nasm编译生成COM文件：

```
nasm ${asm_file}.asm -o ${asm_file}.com
```

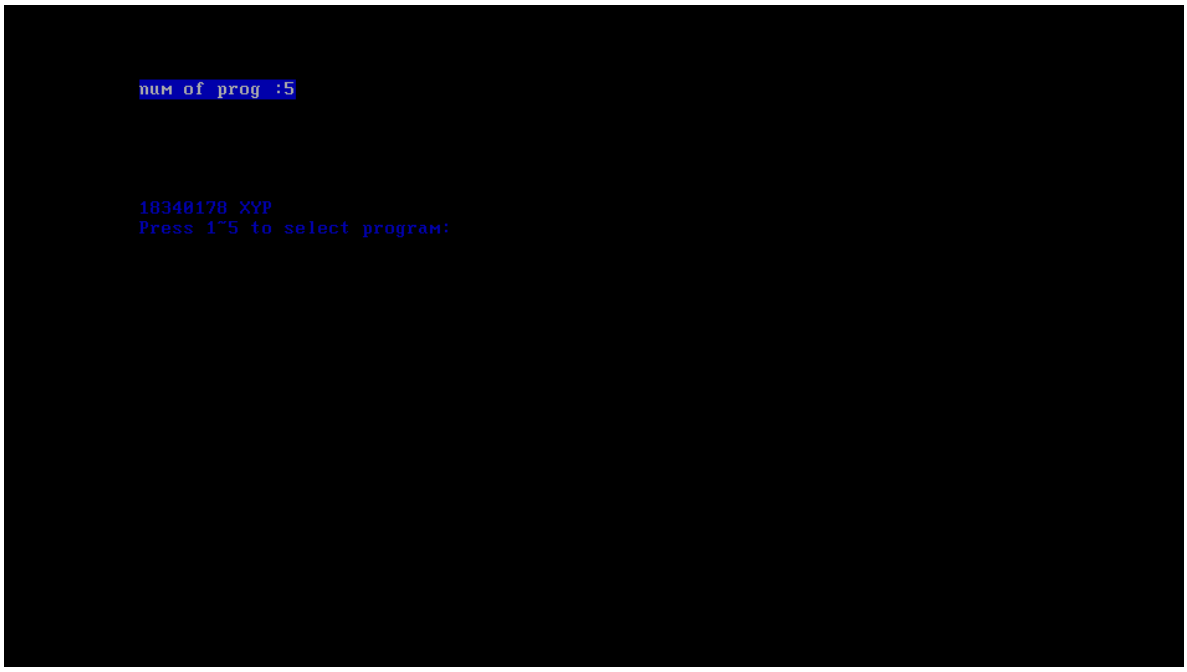
将生成文件写入软盘：

```
cat ${asm_file}.img >> "${output_file}
```

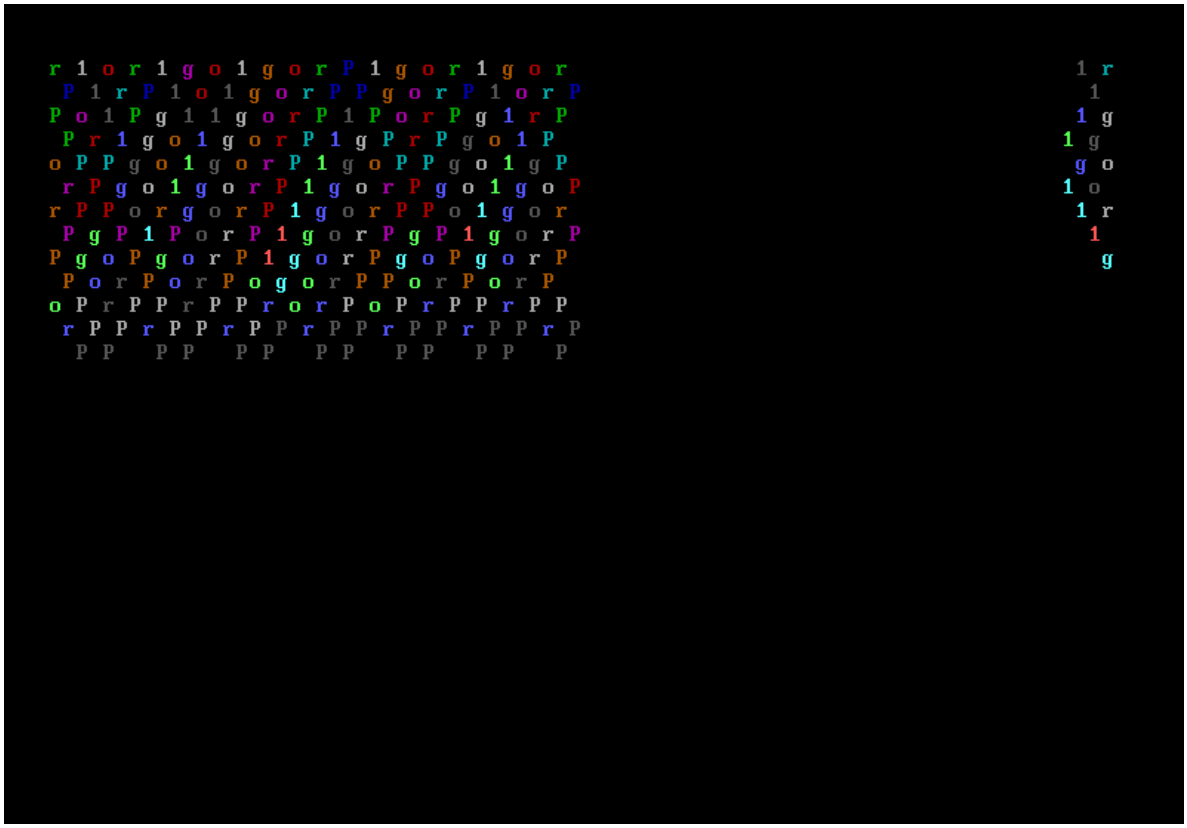
5.运行测试

使用实验一所搭建的虚拟裸机运行生成的img文件进行测试。

Boot

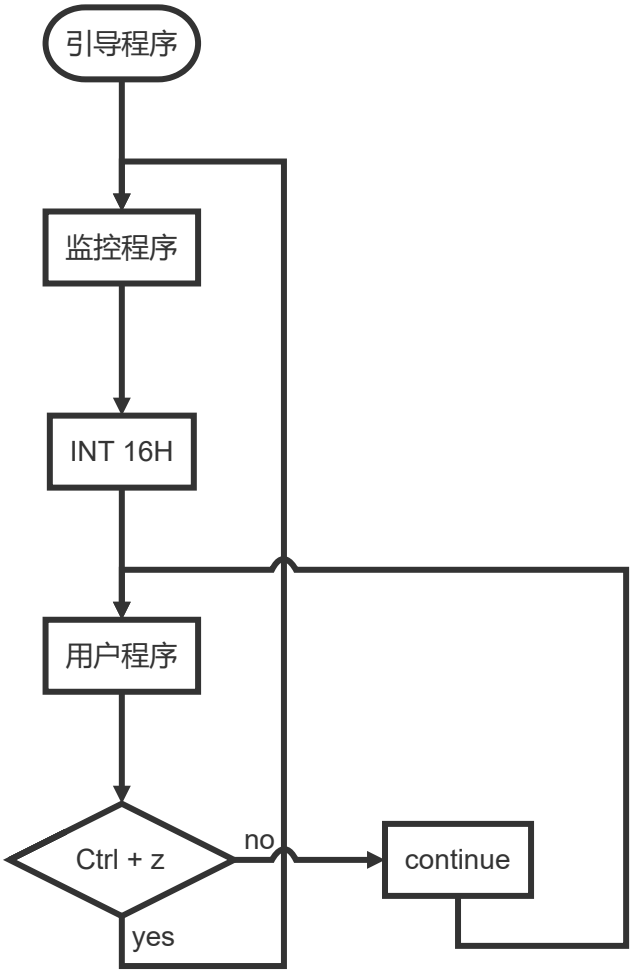


monitor



Prog1

6.流程图



纠错过程

1. jmp可以实现返回监控程序而ret导致用户程序卡死：

最开始实现监控程序 --> 用户程序 时使用call与ret，却发现并不能的到预期效果，用户程序在检测到返回输入指令 即ctrl + z 时回卡死并输入伴有鸣笛声。排查发现是多处使用pusha，popa，导致栈空间溢出，在将不必要的入栈出栈指令删除后解决问题。

2. 输出字符串不符合预期：

由于程序运行实际是加载到内存的结果，有可能缓存区中还有部分字符未及时更新，但其实想不通原因，因为我是分别加载不同用户程序到不同内存空间的，原理上是不应该出现这个问题的，但还是出现了。尝试在运行前加入初始化的语句，结果问题消失（不能算解决问题）。

3. 加载用户程序在错误位置：

原因是read_secotr中调用13H中断需要显式在al给出需要读取的扇区数，查资料时不仔细固定设置为了1，而4个用户程序均由1024字节即2扇区，导致文本缺失。

4. 10H中断显示异常：

10H中断使用前一样需要将gs置于显存0B800H处，否则不能显示出字符串。

实验总结

此次实验是目前耗费时间最多的一次实验，但做完后发现这并不能算很有难度的实验。四个独立的应用程序可以从实验一中稍作修改的到，而监控程序进入用户程序再返回的过程可以用call - ret指令轻松实现。而自己一直在调试的地方也在于此，nasm汇编貌似没有办法直接显示使用cs与ip寄存器，导致自己只能在汇编代码中不断加入输出程序来进行调试，效率极其低下。这也就是没有合理利用工具导致的结果，如Bochs.如果需要再对汇编语言的运行逻辑有更深刻的认识，必须开始学习Bochs调试的相关内容。

Appendix

tl.asm	**用户程序1**
bl.asm	**用户程序3**
tr.asm	**用户程序2**
tl.asm	**用户程序4**
monitor.asm	**监控程序**
boot.asm	**引导程序**
Kernel.img	**整体镜像**