操作系统实验项目

监控程序控制用户程序的执行

郑戈涵　17338233　931252924@qq.com

摘要

本次实验总共完成两个任务：获取计算机硬件系统的控制权和控制用户程序的执行

目录

1 实验目的

1. 了解监控程序执行用户程序的主要工作
2. 了解一种用户程序的格式与运行要求
3. 加深对监控程序概念的理解
4. 掌握加载用户程序方法
5. 掌握几个 BIOS 调用和简单的磁盘空间管理

2 实验要求

1. 知道引导扇区程序实现用户程序加载的意义
2. 掌握 COM/BIN 等一种可执行的用户程序格式与运行要求
3. 将自己实验一的引导扇区程序修改为 3-4 个不同版本的 COM 格式程序，每个程序缩小显示区域，在屏幕特定区域显示，用以测试监控程序，在 1.44MB 软驱映像中存储这些程序。
4. 重写 1.44MB 软驱引导程序，利用 BIOS 调用，实现一个能执行 COM 格式用户程序的监控程序。
5. 设计一种简单命令，实现用命令交互执行在 1.44MB 软驱映像中存储几个用户程序。
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

3 实验内容

3.1 获取计算机硬件系统的控制权

1. 将自己实验一的引导扇区程序修改为一个 COM 格式程序，程序缩小显示区域，在屏幕第一个 1/4 区域显示，显示一些信息后，程序会结束退出，可以在 DOS 中运行。在 1.44MB 软驱映像中制定一个或多个扇区，存储这个用户程序 a。
2. 将自己实验一的引导扇区程序修改为第二、第三、第四个的 COM 格式程序，程序缩小显示区域，在屏幕第二、第三、第四个 1/4 区域显示，在 1.44MB 软驱映像中制定一个或多个扇区，存储用户程序 b、用户程序 c、用户程序 d。

3.2 控制用户程序的执行

1. 重写 1.44MB 软驱引导程序，利用 BIOS 调用，实现一个能执行 COM 格式用户程序的监控程序。程序可以按操作选择，执行一个或几个用户程序。解决加载用户程序和返回监控程序的问题，执行完一个用户程序后，可以执行下一个。
2. 设计一种命令，可以在一个命令中指定某种顺序执行若干个用户程序。可以反复接受命令。

4 实验原理

4.1 BIOS 调用

BIOS 是英文"Basic Input Output System" 的缩略语，直译过来后中文名称就是" 基本输入输出系统"。其实，它是一组固化到计算机内主板上一个 ROM 芯片上的程序，它保存着计算机最重要的基本输入输出的程序、系统设置信息、开机后自检程序和系统自启动程序。其主要功能是为计算机提供最底层的、最直接的硬件设置和控制。

BIOS 芯片中主要存放：

- 自诊断程序：通过读取 CMOSRAM 中的内容识别硬件配置，并对其进行自检和初始化；
- CMOS 设置程序：引导过程中，用特殊热键启动，进行设置后，存入 CMOS RAM 中；
- 系统自举装载程序：在自检成功后将磁盘相对 0 道 0 扇区上的引导程序装入内存，让其运行以装入 DOS 系统；
- 主要 I/O 设备的驱动程序和中断服务：由于 BIOS 直接和系统硬件资源打交道，因此总是针对某一类型的硬件系统，而各种硬件系统又各有不同，所以存在各种不同种类的 BIOS，随着硬件技术的发展，同一种 BIOS 也先后出现了不同的版本，新版本的 BIOS 比起老版本来说，功能更强。

4.2 BIOS 中断例程

BIOS 中断服务程序是微机系统软、硬件之间的一个可编程接口，用于程序软件功能与微机硬件实现的衔接。DOS/Windows 操作系统对软、硬盘、光驱与键盘、显示器等外围设备的管理即建立在系统 BIOS 的基础上。程序员也可以通过对 INT 5、INT 13 等终端的访问直接调用 BIOS 终端例程。

常用的中断命令可以查看附录中的 BIOS 中断向量表（部分）：**[?]**

5 实验过程

这次实验的开放性也很强，为了方便以后的实验，虽然监控程序的功能可以在引导程序中直接实现，我依然把他分开出来，作为一个简陋的内核程序。因此，这次实验需要 5 个步骤，分别是：

1. 设计四个用户程序，内容与第一次实验类似
2. 设计引导程序，将需要的扇区通过 BIOS 加载到内存中
3. 设计监控程序，能够执行加载在内存中的程序
4. 设计命令并对用户程序进行修改
5. 生成镜像，在虚拟机中加载

5.1 用户程序设计

为了理解 bin 和 com 的区别，我写了两个版本的代码。两个版本的用户程序的区别只有开头是否有 org 100h，有的汇编生成后被认为是 com 文件，没有的被认为是 bin。除此之外，

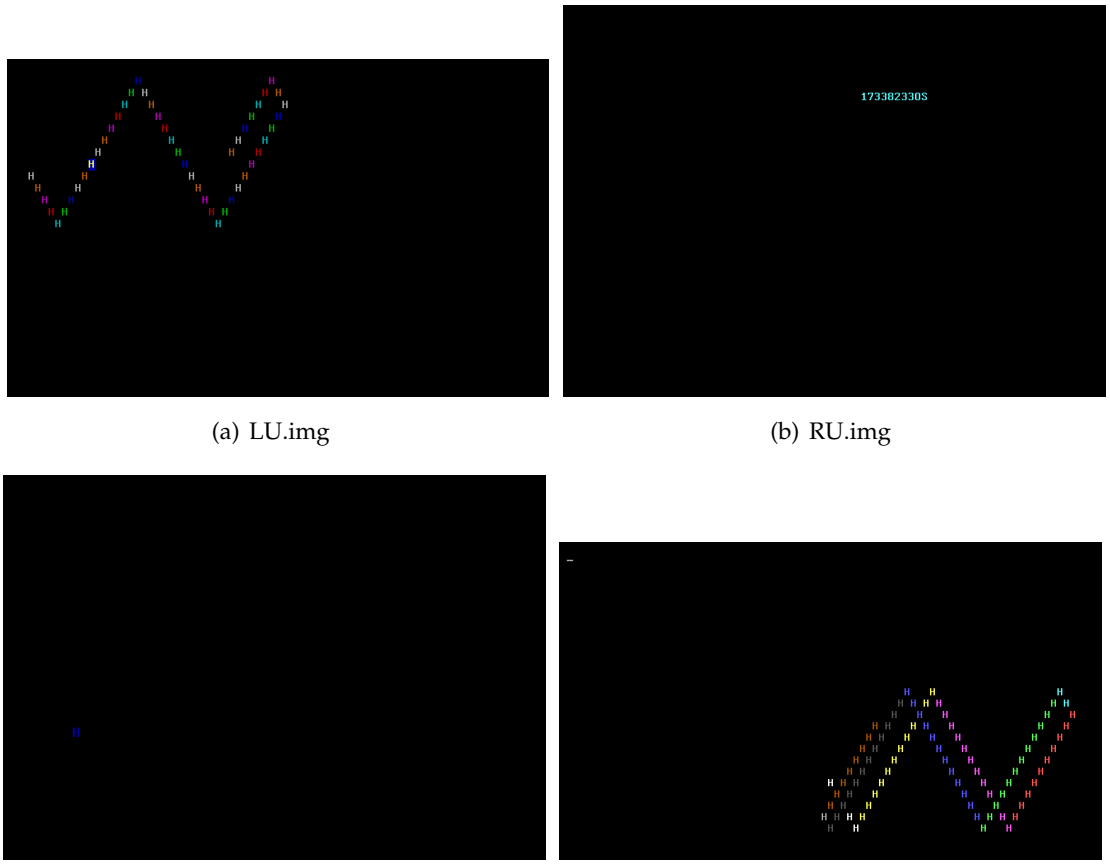
left	字符运动的左边界
top	字符运动的上边界
right	字符运动的右边界
bottom	字符运动的下边界
orix	字符运动的横坐标
oriy	字符运动的纵坐标

代码与第一次的基本一致，但是为了复用，我将一些重复出现的常量放在了头文件里，分别是Dn_Rt,Up_Rt,Up_Lt,Dn_Lt,delay,bigdelay,ddelay。

重复出现的变量在每个程序开头用宏的方式定义在上表中。

剩下的代码与第一次实验基本一致，只是修改了一些显示的特效，比如是否保留轨迹，显示的内容等。以及字符运动的相关参数。四个程序的功能如下：

1. LU.asm: 每一次坐标变化颜色都会变化
2. LD.asm: 一个字符'H' 在运动
3. RU.asm: 字符串"17338233OS" 在运动
4. RD.asm: 字符碰到边界时变色



(a) LU.img (b) RU.img

(c) LD.img (d) RD.img

图 1: 用户程序的运行结果

由于部分程序中代码较多，超过了 512 字节，为方便计算盘块地址，我将四个程序统一改成了 1024 个字节。写好代码后，将四个程序分别编译，并运行，运行结果如上图**??**所示。

5.2 引导程序设计

引导程序的任务是将需要的盘块加载到内存中，然后跳转到监控程序。除此之外，为了让用户知道操作系统正在运行，还需要输出一些语句用于提示。

5.2.1 提示字符串的输出

由于之后的程序中输出需要反复使用，因此根据老师提供的代码，我将用系统调用的输出语句部分定义成宏放在了头文件里。但是要注意的是，老师的代码是直接嵌入在代码中的，直接取出作为宏是不行的，必须在进入时保护现场，离开时恢复。这样才能保证已有的数据不被破坏。这个宏在任何位置展开都不会有问题。

代码如下:

Code 1: 字符串输出宏

```
%macro PRINT 4
   pusha                ; 保护现场
   mov    ax, cs        ; 置其他段寄存器值与CS相同
   mov    ds, ax        ; 数据段
   mov    bp, %1        ; BP=当前串的偏移地址(%1)
   mov    ax, ds        ; ES:BP = 串地址
   mov    es, ax        ; 置ES=DS
   mov    cx, %2        ; CX = 串长（=%2）
   mov    ax, 1301h     ; AH = 13h（功能号）、AL = 01h（光标置于串尾）
   mov    bx, 0007h     ; 页号为0(BH = 0) 黑底白字(BL = 07h)
   mov dh, %3           ; 行号=%3
   mov    dl, %4        ; 列号=%4
   int    10h           ; BIOS的10h功能： 显示一行字符
   popa                 ; 恢复现场
%endmacro
```

输出字符串之后，由于读盘块的时间过短，用户几乎没办法看到显示的内容就被清屏了，因此我设置了一段延时，这段代码来自第一次实验。

代码如下:

Code 2: 延时

```
WaitLoop:
   dec word[count]
   jnz WaitLoop
   mov word[count],bigdelay
   dec word[dcount]
   jnz WaitLoop
```

5.2.2 监控程序和用户程序的加载

加载程序需要用到 BIOS 的系统调用，可以使用 int 13h 来将软盘中的扇区加载到内存中。

代码如下：

Code 3: 加载盘块到内存中

```
LoadKernel:
  ;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, OffsetOfKernel   ;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 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
  ; 用户程序a.com已加载到指定内存区域中
```

为了方便修改，我将 kernal.asm 等文件的偏移量作为 OffsetOfKernel 等宏定义为在头文件。加载其他盘块的代码基本相同，只是修改扇区号和扇区数（4 个用户程序的扇区数为 2）。加载完成后，用 jmp 跳转到监控程序执行。

5.3 监控程序设计

首先设计镜像的结构，也就是确定各个盘块的位置，由于 7c00h 之前的位置不能使用，因此选择之后的位置，并为各个盘块留足空间即可。我的安放位置如下：

Code 4: 偏移量宏

```
OffsetOfKernel  equ   08100h
OffsetOfUserPrg1  equ   0A300h
OffsetOfUserPrg2  equ   0A700h
OffsetOfUserPrg3  equ   0AB00h
OffsetOfUserPrg4  equ   0AF00h
```

监控程序首先清屏，然后输出欢迎语句，清屏使用了 BIOS 的 10 号功能，为了能多次使用而不浪费空间，我将其定义成了子程序。和 print 一样，需要保护现场和恢复现场。

代码如下：

Code 5: 清屏

```
  cls:
  pusha
  mov ax, 0003h
  int 10h          ;清屏
  popa
  ret
```

而接下来两种文件格式便有较大的差别，首先要知道的 com 实际上和 bin 文件没有本质上的区别, 只是开头的 org 会影响整个文件的偏移量, 比如 org 为 100h，会使得整个文件中的 label 都出现 100h 的偏移，而不会影响 ds，所以如果使用 bin，也就是自己决定偏移量，会少很多问题。而使用 com 时，需要注意数据段和代码段地址的计算。

5.3.1 bin 格式

bin 格式非常简单，因为宏对应的就是文件实际所在的物理地址，所以直接跳转不会出现其他问题，接下来通过 BIOS 的 16 号的 01h 调用获得键盘输入，便可以按照用户的需求运行程序代码如下:

Code 6: 获得键盘输入并跳转

```
WaitForKey:
        mov ah,0
        int 16h
        cmp al,'1'
        jz OffsetOfUserPrg1
        cmp al,'2'
        jz OffsetOfUserPrg2
        cmp al,'3'
        jz OffsetOfUserPrg3
        cmp al,'4'
        jz OffsetOfUserPrg4
jmp WaitForKey
```

5.3.2 com 格式

com 格式由于用户程序并不知道自己的物理地址，需要修改段寄存器，包括 cs 和 ds, 而这一工作在 bin 格式的汇编里面也可以完成（没有意义），所以监控程序需要完成段地址的计算和跳转，nasm 汇编提供了 *call far* 指令，可以完成段间转移，该指令完成的工作如下 **[?]**:

1. push CS
2. push IP
3. jmp far ptr 标号

所以接下来需要计算各个文件对应的段基址和偏移地址，由于 org 100h, 偏移地址 ip 是 100h，而段基址通过文件所在物理地址通过以下公式计算得到

$$ds = (OffsetOfUserPrg - 100)/10 \tag{1}$$

理由是段地址左移 4 位加上偏移地址得到真正的物理地址, 即

$$PA(\text{physic address}) = ds << 4 + offset \tag{2}$$

按照镜像文件的设计，数据段的跳转地址如下:

Code 7: 用户程序对应的跳转地址

```
proc1 dw 100h,0xa00
proc2 dw 100h,0xa20
proc3 dw 100h,0xa60
proc4 dw 100h,0xaa0
```

接下来完成代码，获得输入的部分和 bin 格式一样，而这次的跳转是去执行各自的段间转移

Code 8: 获得键盘输入并跳转

```
WaitForKey:
        mov ah,0
        int 16h
        cmp al,'1'
        jz program1
        cmp al,'2'
        jz program2
        cmp al,'3'
        jz program3
        cmp al,'4'
        jz program4
        jmp WaitForKey
```

段间转移代码如下：

Code 9: 获得键盘输入并跳转

```
program1:
        call far [proc1]
        jmp start
program2:
        call far [proc2]
        jmp start
program3:
        call far [proc3]
        jmp start
program4:
        call far [proc4]
        jmp start
```

这样子便完成了监控程序的设计。

5.4 用户程序的修改

核心步骤已经完成，然而这样设计出来的操作系统，用户一旦进入了程序就无法离开。无法按照用户的命令运行程序。因此，一种简单的设计是用户在运行程序时可以通过键盘输入恢复到选择的界面。再选择想要运行的程序，修改内容如下：

1. 在程序的开始时初始化各个变量

2. 显示与程序相关的提示信息，以及离开程序的提示
3. 获得用户的响应，并回到监控程序

5.4.1 程序开始时初始化变量

按照上面的代码，如果在程序执行了一段时间后离开，内存中的数据会保留着被修改过的状态，若重新进入会从上次执行的位置开始，按照我的理解，监控程序每次执行一个程序应该相当于从头开始执行，所以需要初始化。

5.4.2 显示与程序相关的提示信息

显示信息可以使用上面定义好的宏 *PRINT*。

5.4.3 获取用户的响应

在显示字符后获得响应需要用到 BIOS 的 16 号的 01h 调用，此调用与 0h 号调用不同，不阻塞程序的进行，但是获得的是键盘的输入状态。因此用户可以在运行的同时输入。此时没有输入则回到循环开始，有输入则通过 0h 号调用获得输入并比较跳转。我使用的比较的字符是 Esc(27), 各个字符对应的键盘扫描码可以在 BIOS Key Codes中找到。这样用户在程序运行期间的任意时刻按下 Esc 键，都会回到监控程序的界面。

而两种格式返回监控程序的方法也是不同的，com 版本中，由于我使用了 *call far* 指令完成了段间转移，从用户文件返回时需要使用 *retf*。bin 版本的直接跳转返回即可。

此时如果重新将各个文件各种编译生成 img，运行时会发现都是乱码，这是因为单独运行时是在主引导扇区 7c00h, 而文件偏移量不正确，将其改成 7c00h 即可正常运行。

5.5 镜像文件生成

需要的代码已经写好, 接下来将下述代码保存在 run.sh, 下面的是 bin 版本的

代码 10: 生成镜像 (bin):run.sh

```
#!/bin/bash
outfile="os17338233.img"
assembly=("myos1" "kernel" "LU" "LD" "RU" "RD")
rm -f ${outfile}
for asm_file in ${assembly[@]}
do
  nasm ${asm_file}.asm -o ${asm_file}.bin
    cat ${asm_file}.bin >> "${outfile}"
    rm -f ${asm_file}.bin
    echo "[+] ${asm_file} done"
done
echo "[+] ${outfile} generated successfully."
```

下面是 com 版本的，使用了 dd 命令:

代码 11: 生成镜像 (com):run.sh

```
#!/bin/bash
outfile="os17338233.img"
assembly=("myos1" "kernel" "LU" "LD" "RU" "RD")
rm -f ${outfile}
for asm_file in ${assembly[@]}
do
  nasm ${asm_file}.asm -o ${asm_file}.com
done
dd if=myos1.com of=${outfile} bs=512 count=1 2>/dev/null
dd if=kernel.com of=${outfile} bs=512 seek=1 count=1 2>/dev/null
dd if=LU.com of=${outfile} bs=512 seek=2 count=2 2>/dev/null
dd if=LD.com of=${outfile} bs=512 seek=4 count=2 2>/dev/null
dd if=RU.com of=${outfile} bs=512 seek=6 count=2 2>/dev/null
dd if=RD.com of=${outfile} bs=512 seek=8 count=2 2>/dev/null
for asm_file in ${assembly[@]}
do
  rm -f ${asm_file}.com
    echo "[+] ${asm_file} done"
done

echo "[+] ${outfile} generated successfully."
```

在命令行中执行即可生成镜像 (.img) 文件，在 VMware 中加载该镜像，查看结果。到此，实验结束。

6 程序使用说明

6.1 实验环境

6.1.1 VMware Workstation 15

VMware Workstation 是 VMware 公司推出的一款桌面虚拟计算软件，具有 Windows、Linux 版本。此软件可以提供虚拟机功能，使计算机可以同时运行多个不同操作系统。此次实验中用于运行镜像文件。

6.1.2 nasm 2.13.02

Netwide Assembler 是一款基于英特尔 x86 架构的汇编与反汇编工具。在此次实验中用生成二进制文件 (.bin)

6.1.3 wsl

本次实验中用到了 shell 文件以批处理源代码生成镜像，整合于 windows 系统的 wsl 能够很方便的处理在 windows 系统上生成的文件。

6.1.4 VSCode 1.44.2

vscode 能够方便的使用 wsl 和 latex 的 pdf 生成功能。加载虚拟机以外的所有任务都可以通过 vscode 完成。

6.2 编译方法

6.2.1 系统要求

生成镜像文件时，可以使用 linux 操作系统或者带 wsl 的 windows 系统。

6.2.2 编译过程与参数

在源代码目录下使用 wsl，执行下列代码即可得到镜像文件 os17338233.img。

代码 12: 生成镜像

```
./run.sh
```

6.3 运行与演示

两个版本的运行结果是一样的，下面以 bin 版本的为例在虚拟机软件 (此实验中为 VMware Workstation) 中创建虚拟机，将镜像作为软盘载入，设置随虚拟机启动后启动虚拟机即可。

运行结果如图 **??** 所示。

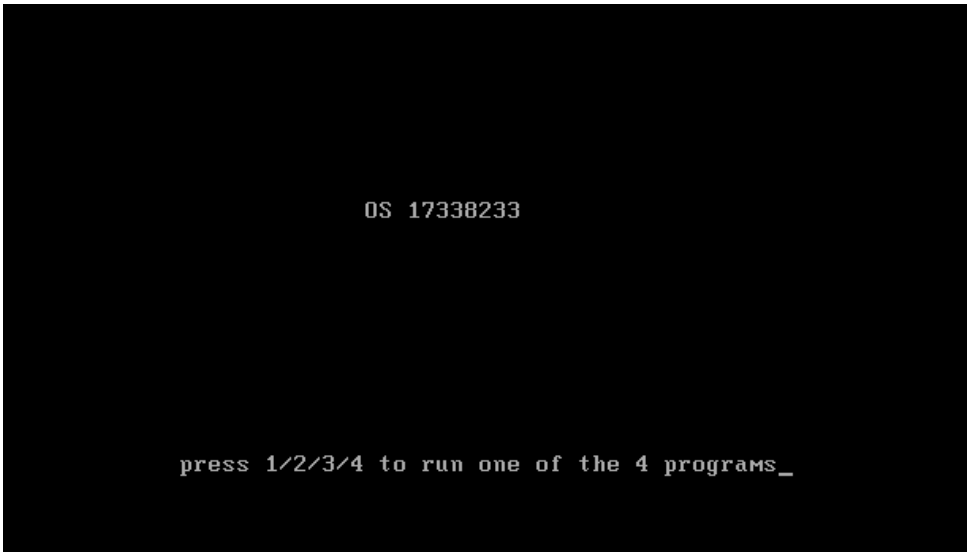


图 2: 运行过程截图

具体的演示过程可以查看已录制好的 OSdemo.mkv

7 总结与讨论

7.1 特色, 不足与改进

本程序通过使用宏简化代码量，并使得常量易于变更，测试时更加方便。不足之处是用户程序比较简单，命令比较单调。本实验我尝试着生成了两种不同格式的可执行文件，com 版本由于计算段基址不能直接使用宏所以都为计算后手动填入，在测试时修改较为麻烦。修改程序时容易出现地址不一致的问题。

7.2 收获

通过本次实验，我复习了 x86 汇编，回顾了子程序的调用方式（call, 入栈和出栈），了解了带参数的宏的使用以及头文件的包含，宏 *PRINT* 以及各个块的偏移量在代码中反复出现，非常实用。我也对操作系统的启动过程，还有 BIOS 的调用，尤其是阻塞方式和非阻塞方式获取键盘输入有了一定的了解。尝试编写两种格式的汇编时，我也对段间转移，物理地址的计算，汇编代码中标签的偏移量，org 的意义有了更深入的理解。同时，在本报告的编写过程中，我也练习了利用 LaTeX 编写文档的能力。[**?**, **?**]

7.3 感想

作为操作系统的第二个实验，代码量和第一次相比有较大的增加，并且完成的功能也更加复杂，为了理清项目的结构我在实验开始时花了较长时间查询相关资料。

而实验过程中也遇到了不少问题，比如修改了字符串显示的位置有时候会导致看不到显示的内容，或者使用的段寄存器不同 (gs 和 es) 也会导致输出乱码。前者后来并不能复现。后者是因为在不同文件里给段寄存器赋了不同的值导致数据偏移量错误，说明段寄存器的功能要预先分配好，否则会出现冲突，这些问题刚出现时我完全不知道原因，复习了汇编之后才理解，说明写操作系统时，汇编能力是基础。

我遇到的最大的问题则是四个用户程序顺序的错乱，在完成 bin 版本的实验中，由于 shell 文件中我没有刻意安排各个镜像文件放置的位置，运行时输入数字不仅并没有按照我的期望运行。虚拟机还发出了警报声并退出，从中我才明白了对 bin 格式来说物理位置的重要性，将文件的物理地址与代码中的地址隔离开是非常有必要的，使用 com 格式就不会出现上述问题，只是运行的顺序有变化。

此外，这次实验由于子过程较多，我发现同样的调用结果，可以用非常多不同的指令，比如 bin 版本中 *kernel.asm* 中跳转到其他段的部分，如果把用户程序当成子过程，可以这样改:

Code 13: 获得键盘输入并跳转

```
1 WaitForKey:
2       mov ah,0
3       int 16h
4       cmp al,'1'
5       jz program1
6       cmp al,'2'
7       jz program2
```

```
        cmp al,'3'
        jz program3
        cmp al,'4'
        jz program4
        jmp WaitForKey

program1:
        call OffsetOfUserPrg1
        jmp start
program2:
        call OffsetOfUserPrg2
        jmp start
program3:
        call OffsetOfUserPrg3
        jmp start
program4:
        call OffsetOfUserPrg4
        jmp start
```

经过测试，这样得到的结果和上面的一样，看似更加规整，但实际上和 [??] 完成了相同的工作，却使用了更多的空间，在内存大小有限的情况下，还是更需要简洁有力的代码。因此我认为不必要使用 call 的场合，用跳转指令是比较合适的。

这次遇到很多问题都花了不少时间才解决，因为我再使用 bochs 等调试工具时对命令不够熟悉，导致调试时并不能找到代码的问题，要希望在下次实验中我能够更高效的调试程序，比如使用 bochs。

References

1. Leslie Lamport, *LaTeX: a document preparation system*, Addison Wesley, Massachusetts, 2nd edition, 1994.
2. Contributors to Wikibooks, *LaTeX Bibliography Management*. Wikibooks, 2019., en.wikibooks.org/wiki/LaTeX/Bibliography_Management.
3. BIOS 中断调用 *Wikipedia, The Free Encyclopedia* 2019., https://zh.wikipedia.org/wiki/BIOS
4. 汇编语言入门：CALL 和 RET 指令（一）鸾林居士, 2018., https://blog.csdn.net/abc$_1$2366/*article*/*details*/79774530

8 附录：BIOS 中断向量表

表 1: BIOS 中断向量表

中断	描述
INT 10h	显示服务 - 由 BIOS 或操作系统设定以供软件调用。 AH=00h 设定显示模式 AH=01h 设定游标形态 AH=02h 设置光标位置 AH=03h 获取光标位置与形态 AH=04h 获取光标位置 AH=05h 设置显示页 AH=06h 清除或滚动栏画面 (上) AH=07h 清除或滚动栏画面 (下) AH=08h 读取游标处字符与属性 AH=09h 更改游标处字符与属性 AH=0Ah 更改游标处字符 AH=0Bh 设定边界颜色 AH=0Eh 在 TTY 模式下写字符 AH=0Fh 获取当前显示模式 AH=13h 写字符串
INT 11h	返回设备列表。
INT 12h	获取常规内存容量。
INT 13h	低端磁盘服务。 AH=00h 复位磁盘驱动器。 AH=01h 检查磁盘驱动器状态。 AH=02h 读扇区。 AH=03h 写扇区。 AH=04h 校验扇区。 AH=05h 格式化磁道。 AH=08h 获取驱动器参数。 AH=09h 初始化硬盘驱动器参数。 AH=0Ch 寻道。
INT 16h	键盘通信例程。 AH=00h 读字符。 AH=01h 读输入状态。 AH=02h 读 Shift 键（修改键）状态。 AH=10h 读字符（增强版）。 AH=11h 读输入状态（增强版）。 AH=12h 读 Shift 键（修改键）状态（增强版）。
INT 19h	加电自检之后加载操作系统。