16 级计科 7 班: 操作系统原理实验 #2

Due on Monday, March 19, 2018

凌应标 周一 *9-10* 节

颜彬

16337269

Content

1 实验目的

设计和完善更多的有输出的用户程序。加深对 16 位 x86 汇编的理解。加深对汇编程序中 “寻址” 的理解，例如 jmp near 和 jmp far 的区别, 和 org 指令的理解。

理解 bios 中断，掌握基础的 bios 中断的运用，特别是键盘输入中断，字符串输出中断和扇区载入中断。修改并完成控制程序，使其识别并响应键盘按下事件。

理解程序在磁盘和内存中存储方式的不同点。掌握将用户程序载入内存的操作。

2 实验要求

2.1 拓展用户程序

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。

2.2 组织存储方式

自行组织映像盘的空间, 以存放四个用户可执行程序。自行组织内存的空间，以排列控制程序和用户程序等。

2.3 响应按键事件

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

3 实验方案

3.1 基础原理

3.1.1 NASM 宏

NASM 具有极其强大的宏功能。NASM 宏允许程序在编译时成文本替换、代码生成、表达式运算、引入其他文件代码等强大的工作。

本实验要求产生四个及以上的用户程序。本项目利用宏的灵活性，通过调整编译指令，在简单修改源代码的基础上，产生了 6 个用户程序。该 6 个用户程序分别在屏幕的左上角、上方、右上角、左下角、下方、右下角运动。实现如代码1, 编译方法见4.2 节的代码6。

3.1.2 独立的全局函数空间

为了增加代码的可拓展性和可组织性,本项目特地增加了 “全局函数空间”。即在内存 0xA100 至 0xA300 的空间范围内，放置本项目常用的函数的实现，如清屏和打印字符串等。0xA100 地址起始处的若干字节将保留用作索引信息。其他程序只需要检查 0xA100 之后的若干字节，即可知道目标函数的物理地址，并采用 call [reg] 的方式跳转并执行。具体实现方法4.1.3 节的代码3

3.1.3 组织存储方式

真正的磁盘由若干个盘片堆叠而成。每个盘片都有两个磁头，故经常用磁头从上到下的编号来代指盘面的编号。每个磁头可以共同步进或步退，在磁盘高速运转时，磁头的每个运行轨迹对应一个磁道。将磁道继续细分，每个磁道可分为若干个扇区，通常分为 63 个，从 1 编号至 63. 每个扇区 512 字节。

从实验一可知，第一个扇区为引导扇区，在裸机开机时会被自动加载入内存并尝试引导。本项目还有额外的 6 个用户程序和 1 个独立的全局函数程序（见 4.1.3节)。在本实现中，全局函数程序将占据虚拟软盘的第二个扇区。其余的每个用户程序将单独占据一个扇区，并从第三个扇区开始，依次排列。即将第 i 个用户程序放置在虚拟软盘的第 i + 2 个扇区内。写入虚拟软盘的指令实现了这些程序的存放方式。具体指令见4.2节代码7.

3.1.4 ORG 伪指令

由于本项目包含了多份代码文件，有多处跳转和寻址，故理解 ORG 伪指令显得尤为重要。

在一份汇编代码中，所有的 label 在编译期都会转化为相对于程序开头的偏移量。即编译器会把程序的开头作为基准，记开头的地址为 0x0000，并以此计算所有 label 的偏移量。需要注意的是，ORG 并不影响程序最终放在内存的什么位置。程序在内存中的存放是由操作系统完成的，操作系统的调度对于程序而言是 “透明” 的。

假设段寄存器都指向 0x0000 且一直保持不变。假设程序实际被装载入内存 0xA300 处。在程序实际执行时，所有的寻址操作都会跳转到无效地址。这是因为程序永远以以 0x0000 为基准，加上偏移量寻找目标。程序寻找地址的方式是

$$\text{phy\_addr} = \text{base\_addr} \ll 1 + \text{offset}$$

而此时 base_addr 是 0.
ORG 伪指令可以解决这个寻址错误的问题。假设程序员提前知道了自己的程序会被装载入 0xA300 处。ORG A300 伪指令做的事情是，在所有的 label 偏移量计算完成后，再额外加上 0xA300。此时寻址方式变成了

$$\text{phy\_addr} = \text{base\_addr} \ll 1 + \text{offset}'$$

其中

$$\text{offset}' = \text{offset} + \text{0xA300}$$

. 这些操作都由汇编器在编译期间完成。

解决这个寻址错误的问题还有另外一个方法。同样设程序被加载入内存的 0xA300 处。只需将段寄存器设置为

$$\text{segment} = \text{0xa300} \div \text{0x10H} = \text{0x0a30}$$

3.1.5 bios 中断

bios 提供了若干基本的中断供程序调用。其中包括字符（串）输出中断，键盘中断和扇区载入中断等。在程序中，只需要将正确的参数置于相应的寄存器中，并执行 int <num> 即可直接调用相应的中断。

3.1.6 程序寻址

由于本项目除了引导程序外，还具有数个其他程序，代码段间的跳转和数据的寻址显得尤其重要。在 x86 实模式下，寻址采用的方式是

$$\text{addr} = \text{segment} \times 16 + \text{offset}$$

其中，在十六进制下又等价为

$$\text{addr} = \text{segment} \ll 1 + \text{offset}$$

.

值得注意的是，jmp near 和 jmp far 将具有不同的行为。前者不会改变段寄存器，而后者会将段寄存器修改为目标位置的段基地址。在本项目中，所有跳转都采用 jmp near 的方式。

在本项目中，org 的使用尤其关键。它影响了所有数据、函数的寻址。

3.1.7 键盘响应

对按键的响应主要通过 16H 系统中断完成。本项目主要使用 16 号中断的 00H 功能和 01H 功能。

01H 功能会检查键盘缓冲区的内容。若缓冲区为空，ax 将保持不变。若缓冲区中按键缓冲，则 ah 会被修改，而 al 将存储按下键的 ascii 码。

00H 的功能和 01H 十分相似。两者最大的不同在于对缓冲区的处理。01H 功能调用在检查缓冲区后，会保留缓冲区不变。这意味着，若缓冲区不为空，连续的多次使用 01H 号功能将得到相同的返回内容。而 00H 则不同，它在检查缓冲区后，若缓冲区不为空，则会将缓冲区的头部取出（相当于队列的出队操作）；若缓冲区为空，程序会被阻塞直到键盘被按下。

关于按键响应的详细使用，见4.1.2节。

3.2 实验环境

3.2.1 系统与虚拟机

- 操作系统
  本实验在 Linux 下完成。采用 Ubuntu 16.04
- 虚拟机
  bochs. 它是一款开源且跨平台的 IA-32 模拟器。

3.2.2 相关工具、指令

- 汇编器
  NASM. NASM 是一个轻量级的、模块化的 80x86 和 x86-64 汇编器。它的语法与 Intel 原语法十分相似，但更加简洁和易读。它对宏有十分强大的支持。
- 镜像文件产生工具
  bximage. 该命令允许生成指定大小的软件镜像。
- 二进制写入命令
  dd. dd 允许指定源文件和目标文件，将源文件的二进制比特写入目标文件中的指定位置。

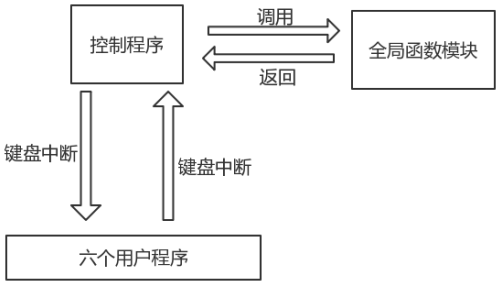

图 1: 程序间的关联方式

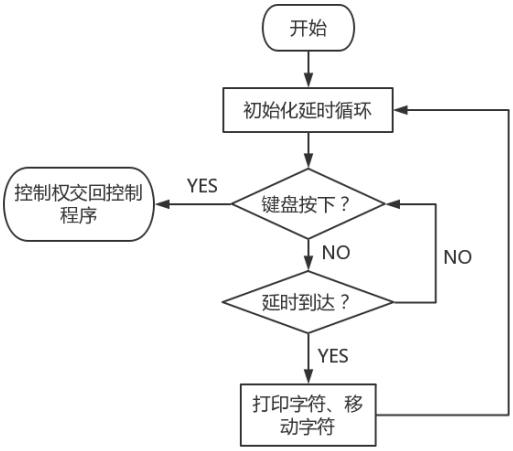

图 2: 用户程序流程图

- 二进制文件查看命令
  xxd. xxd 允许将二进制文件中的内容按地址顺序依次输出，可读性强

3.3 程序流程

图1展示了程序整体间的关联方式。控制程序在获得处理器控制权后，立即将全局函数模块载入内存中。在清屏和输出相关的提示信息后，等待键盘按下。

每当键盘被按下，控制程序会判断按下的是哪个键，并把相应的用户程序载入内存中，随后运行用户程序。在用户程序的运行期间，若用户程序检测到键盘按下，它会先把控制权返回给控制程序，由控制程序统一处理。而控制程序使用到的所有函数，由全局函数模块统一管理。

图2展示了用户程序的流程图。该流程与实验一的流程图十分相似。不同之处在于，每个循环内，用户程序需要检查键盘是否被按下。若键盘被按下，需要将控制权返回给控制程序。

4 实验过程

4.1 程序模块及相关说明

4.1.1 NASM 宏

为了产生 6 个互不相同的用户程序，本代码充分利用了 NASM 的宏特性。如代码1所示。

首先，若编译时带有 DEBUG 选项，则会使 delay 被定义为 1000, 否则 delay 为 50000。以这个方式定义 delay 变量是由于 bochs 虚拟机的性能较差，若 delay 的值较大会导致程序运行速度过低，查错变得更难。但其他虚拟机的运行速度比 bochs 快，需要更大的 delay 才能让程序的动画肉眼可见。故这里根据是否定义了 DEBUG 宏而给出了两种不同的延时。通过简单地修改编译指令，可以使得编译后的程序适应多种机型（和虚拟机）。

代码 1: 用户程序中的部分宏代码

```
; in file stone.asm

%ifdef DEBUG
    delay equ 1000
%else
    delay equ 50000
%endif

%ifdef UL
    mycode equ 'q'
    DownSideBoundary equ 12
    UpSideBoundary equ 4
    LeftSideBoundary equ 1
    RightSideBoundary equ 23
    _x equ 1
    _y equ 10
    org 0A300H
    DIRECTION equ Up_Lt
%elifdef UP
    ; ...
    ; ...
%endif
```

随后，根据程序的六种不同的位置（如左上、右上、左下等），定义了 6 个不同的宏。通过在编译时指出不同的宏，程序将被编译出 6 种不同的版本。如此即可在几乎不修改代码的前提下产生众多相似程序。这六个不同的宏中，每个宏各自定义了符号的起始坐标 x 和 y，上下左右的四个边界的坐标 [(Down)|(Up)|(Left)|(Right)]SideBound, 符号的初始运行方向和程序的偏移地址等。

程序的编译方法对程序的产生具有重大影响。编译命令见4.2节的代码7 。

宏中 org 指令的解释见3.1.4节。

4.1.2 键盘响应

对键盘的响应和中断的使用见3.1.7节的介绍。本节主要对程序代码中的相关部分做解释。

代码2介绍了用户程序中的按键响应方法。

用户程序的主体部分是一个延时循环。每次循环时，程序都会跳转到 test_key_press 过程. 程序首先检查是否有键按下，若没有，则跳转到下一次的延时循环。若有，程序再检测按下的键是否合法，若不合法，同样跳转到下一次循环。若按键合法，则程序会调用一个函数，并将控制权交给控制程序。在进入函数之前，程序首先会做一些准备工作，例如向寄存器中存入传递的参数等。随后，程序将 sys_base 的值置入 bx 中。sys_base 指向一系列常用函数的 “地址表” 的表头。其中按键响应函数的地址在地址表的偏移量 0AH 处。故 call [bx + 0AH] 指令会直接调用键盘响应函数。

有关全局函数地址表和全局函数的问题，见代码3

代码 2: 用户程序中对键盘按下的响应

```
; in file stone.asm
test_key_press:
    xor ax, ax
    mov ah, 1
    int 16H
    cmp ah, 1
    jz loop1.Loop ; if no key is pressed, jump to delay loop

    xor ax, ax
    int 16H
    cmp al, mycode
    jz loop1.Loop ; if pressed key stand for the program itself
                  ; then just ignore it
    ; key pressed

    ; ...
    ; some set-ups
    ; ...

    mov bx, sys_base
    jmp [bx + 0AH] ; jump to a key-press handler
```

4.1.3 全局函数

为了减少控制模块的代码体积，减少各个模块间的耦合度，以及增加可拓展性，本实验将各个程序间共同需要使用的函数抽取出来。当需要调用这些共享函数时，只需要查阅 0xA100 地址起始处的一个表，找到函数入口地址即可。

如代码3所示，该代码段首先声明了 org 0A100H，这是因为控制函数会把该其载入到内存 0xA100 处。紧接着代码段声明了若干个单字长的空间，每个空间存储着对应函数的入口地址。例如 _show_at_call (0xA104H 处) 这一 16 位空间存储着函数 show_at_call 的入口的地址。每个地址空间声明为 16 位长 (dw) 而不是其他的长度是因为，8 位太短，很可能不足以表示部分函数的地址; 不声明为 32 位或以上的原因是，每个段的段长最多为 16 位，在不改变段寄存器的前提下，16 位刚刚好足以表示所有段内函数的地址。

在该代码段中，loader_address 和 loader_back_addres 是两个比较特殊的表项。他们分别指向控制程序的首地址和控制程序中按键响应函数的首地址。由于控制程序的首地址难以在编译器直接确定，故控制程序在载入内存后，会立即修改这两个表项的值，使其指向正确的地址。

使用这些全局函数的方法很简单。由于这些函数会被载入到内存地址 0xA100 处，故只需先将 bx 的值赋为 0xA100, 并采用 call near [bx + offset] 即可。

4.1.4 扇区载入

控制程序需要将其他的程序载入到内存中。在本项目中，控制程序还要尽早将全局函数加载入内存，以便控制程序（和用户程序）调用其中的函数。

代码 3: 全局函数和其地址表的定义

```
; in file Utilities/utility.asm
org 0A100H
loader_address dw 0xF0F0 ; 0xA100H        0xA100H + 0
loader_back_address dw 0xF0F0 ; 0xA102H        0xA100H + 2
_show_at_call dw show_at_call ; 0xA104H        0xA100H + 4
; ...
; more items here
; ...
show_at_call:
    ; bp point to str address
    ; cx contents the length
    mov ah, 0EH
    push bx
    mov bl, 0
    mov al, '@'
    int 10H
    pop bx
    jmp [loader_back_address]

; ...
; more functions here
; ...
```

同时，当对应的按键被按下时，控制程序还应载入相应的用户程序到相应的内存地址，并跳转到该内存地址以执行用户程序。

控制程序的代码见代码4.

4.2 编译方法

如代码5所示，编译本个项目的方法已经封装成脚本 build.sh。该脚本会递归地调用 Utilities/build.sh 和 userCodes/build.sh，完成这些组件的编译过程。

其中 userCodes/build.sh 的内容如代码6所示。该代码在使用 nasm 六次编译文件时，都给出了-D DEBUG 的选项。该选项的介绍见4.1.1节. 这六个文件，在编译时各给出了 UL,UP,UR,DL,DN,DR 六个选项，分别代表从左上到右下的六个不同方向。在编译完成后。这六份代码中的字符'A' 会以不同的位置和角度在各自的区域内反射。

其中将二进制代码写入虚拟软盘的代码如代码7所示。
boot.bin 作为引导程序和控制程序放在第一个扇区。utility.bin 作为全局函数放在第二个扇区。其余用户程序依次放在连续的扇区中。sh 脚本采用"$((expr))" 的方式计算各个 bin 文件的放置位置。

4.3 运行方法

本项目的运行方法与项目一的运行方法一致。

代码 4: 控制程序中的扇区载入部分

```
; in file loader.asm

; ...
; after a key has pressed
; ...
;   mov cl, 3
;   mov bx, 0xA300
    call load
    jmp bx


    load:
    ; cl the nth sector to load
    ; bx the base address to put the code
.body:
    mov ax, 0
    mov es, ax ; which segment to load
    mov ah, 2 ; function number
    mov al, 1 ; load one sector
    mov dx, 0 ; dl = 0 for floppy disk, dh = 0 for 0 head
    mov ch, 0
    int 13H ;
    ret
```

代码 5: 编译程序代码的方法

```
$ sh build.sh
```

代码 6: userCodes/build.sh 文件内容

```
set -x
nasm stone.asm -o ul.bin -d DEBUG -d UL
nasm stone.asm -o up.bin -d DEBUG -d UP
nasm stone.asm -o ur.bin -d DEBUG -d UR
nasm stone.asm -o dl.bin -d DEBUG -d DL
nasm stone.asm -o dn.bin -d DEBUG -d DN
nasm stone.asm -o dr.bin -d DEBUG -d DR
```

代码 7: 向虚拟软盘内写入比特的部分代码

```
dd if=loader.bin of=boot.img conv=notrunc
dd if=Utilities/utility.bin of=boot.img conv=notrunc \
    oflag=seek_bytes seek="$((512*1))"
dd if=userCodes/ul.bin of=boot.img conv=notrunc oflag=seek_bytes seek="$((512*2))"
dd if=userCodes/up.bin of=boot.img conv=notrunc oflag=seek_bytes seek="$((512*3))"
// ...
// ...
dd if=userCodes/dr.bin of=boot.img conv=notrunc oflag=seek_bytes seek="$((512*7))"
```

运行的方法同样被封装成一个脚本，直接运行即可。见附录B的代码8. 其中文件 run.sh 的内容见附录B的代码9. 运行虚拟机 bochs 的配置文件见附录B的代码10

5 实验结果

程序开始运行时，会清屏并显示一个欢迎页。欢迎页上有基本的操作方法和本人的学号姓名。如图3所示。

按照欢迎页的指引，依次按下 Q,W,E,A,S,D 六个键后，六个用户程序将依次运行。此时程序的运行图如图4所示。

按下 L 后将执行清屏。清屏并不会将本人的学号和姓名清空。只会将这六个用户程序的画面清空。故清屏后的屏幕显示效果与图3一致。

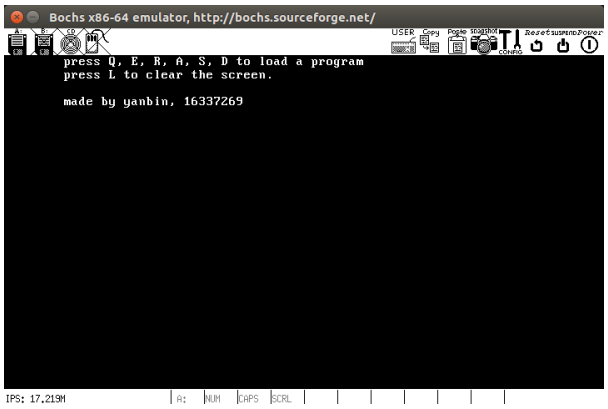

图 3: 开始程序时的欢迎页和清屏后的页面

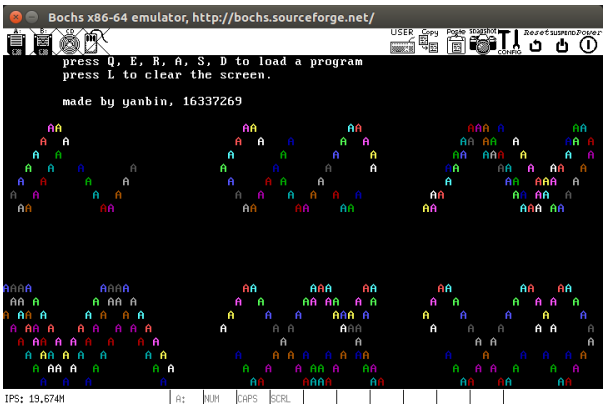

图 4: 六个用户程序的运行图

6 实验总结

6.1 心得体会

这次实验，我经历了十分顺利的过程，也遇到了很多坎坷。

顺利的地方在于，我对 x86 和 nasm 的一些特性比较熟悉，在编程之前，我已经把大概的想法在心里过了几遍。所以代码的实现过程很短。而且实现完后几乎无需怎么修改就能正常地运行了。

但我在这次项目中遇到了一个十分严重的 bug。我用了许多 bochs 自带的 debugger 功能，花了一天多的所有课余时间，才把这个 bug 最终解决。**bug 的解决过程见6.2节的心路历程，此处不再赘述**。由于这个 bug 隐藏得比较深，故在这次 debug 的过程中，我花了很多时间在内存级别上观察数据和跟踪程序的跳转。这个过程虽然繁琐，却让我对大小端、数据寻址和指令跳转地址等较为繁琐的知识点有了直观的感觉。通过手动计算数据的偏移量和物理地址，我对汇编中寻址的操作又有了新的认识。

在这次实验中，我把一些常用的函数单独地抽取出来，存放到虚拟软盘的一个扇区中，并加载到内存的特定位置。因为我考虑到，在以后的操作系统实验中很可能会需要相关的操作，可以利用这次实验先熟悉这个做法。共享的函数不应该在每个用户程序中都有一份单独的拷贝，这样太浪费内存空间。把函数单独取出，并供所有用户程序共享才是正确的做法。这次实验虽然没有实现任何的内存保护（没有进入保护模式），但我相信这个实验可以为以后的保护模式奠定基础。

下一个实验将是 C 和汇编的混合编程。汇编能给我们操作底层的可能，但用汇编实现数据结构和算法未免深感桎梏。希望在完成了下个实验后，能把握好 C 语言这个利器，创造更多的可能。

6.2 遇到的 BUG

6.2.1 心路历程

但这次项目，我依旧遇到了几个严重的 bug。这个 bug 会导致程序运行了一段时间后，发生异常，包括卡死、卡退、屏幕变花等。其中卡死是最最常发生。其余的情况只遇见过一两次。这些 bug 的严重之处在于，它们在程序的每次运行时都会发生，但我却很难找到一种稳定复现 bug 的方法。也就是，我很难找到一种简便的操作，使得程序 “一定在这一步后卡死”。有时程序在一开始就会被卡死，有时程序能正常运行一分多种的时间，却又突然在某一个操作后卡死。我唯一能确定的是，卡死一定是在按键按下时发生，不按键盘是不会卡死的。

在我被这个 bug 折磨了一整天后，我依旧毫无进展。我甚至无法定位到 bug 的具体位置。因为每次出状况时程序所在的代码段都不一样！甚至程序会运行内存中其他位置的代码段，甚至程序会执行非法代码！由于我的程序中出现了大量的跳转（我实现一些比较复杂的跳转逻辑，因为以后的实验可能会用到), 我很难反向追踪到，到底是哪一条跳转指令将我的程序带到各种非法的内存处。

在王永锋同学的建议下，我换了一个虚拟机 qemu。意外的是，在 qemu 虚拟机上运行，这个 bug 居然一次都没有出现。于是接下来的半天里我都把这个 bug 归咎到 bochs 虚拟机的头上。但我依旧不死心。尤其是我查阅到 bochs 的最后一次更新在 2018 年。这意味着 bochs 是一个被长期维护的虚拟机。这种 “活跃” 的开源软件绝对不可能有这么明显的 bug。所以我又开始认定了是我自己的问题。

在 debug 的期间，我用尽了各种方法。我甚至让 bochs 开启了 trace-mem on 指令。该指令会将所有的内存修改和访问输出到终端。但我依旧一无所获，因为程序在运行时修改和读取内存是极其常见的事情，我的终端很快被海量的输出信息给占满了，我根本找不到我需要的信息。在我持续的努力下，我偶然发现了程序的代码段被改写了。我眼睁睁地看着代码段 0xA4B0 中的值在调用中断的前后发生了变化。这让我更加困惑了。于是我用 watch write 0xA480 指令监控内存，反复将程序跑了几次，发现这个内存被改写的情况时有发生（但却不是必然发生），而且一定是在调用中断时被改写内存。

经过了详细的思考和同学间的讨论。我逐渐怀疑是发生了缓冲区溢出。因为我发现自己无意间在程序

开头写下了 mov sp, ax 这样的代码。我的程序没有大量用到栈，所以我没有特地为栈预留很大的空间。我预估着若干个 push 和 call 等还是安全的。但我突然想到，很可能是调用中断时 INT 指令依旧利用了我当前的用户栈，发生了栈溢出，先是改写了我的数据段，最后甚至改写了代码段。**本次实验观察到的栈溢出现象十分有趣，可见附录B的代码11**第 5 行和第 14 行。

我修改了代码，把栈指针扔到了一个很远的地方，预留了一大块空间。随后我突然发现我的一个数据的数据大小声明错误了。用户程序中记录字符 'A' 坐标的 x 变量，我声明成了 db，但应为 dw。我迅速将这个错误也改了过来，于是程序能完美地运行，不再有任何 bug 了。

后来我经过详细分析，逐渐明白了这种现象的原因。由于我的程序中，从低地址到高地址依次排列着代码段，数据段和未使用空间（栈指针指向其末尾）。故缓冲区溢出时，首先是部分数据被改写了。数据段中不是所有数据都十分重要且会被立即访问到，故少量的溢出不会立即改变程序的行为。栈溢出进一步进行，甚至意外地改写了我的程序代码。在程序继续执行的过程中，执行到了被覆写的内存，于是出现了不同程度的异常。这些异常包括，运行到错误的 jump $ 指令，运行到非法指令，偶然跳转到 0x0000-0x0100 的非法内存等。程序不会在栈溢出的瞬间出现问题，而是在其后的一小段时间后才出现异常，且异常的行为各种各样。这就是造成我的程序 debug 如此困难的主要原因。同时，数据长度声明错误带了的影响，也与栈溢出类似。它不会立即造成程序的错误，而是在某个特别的时机使程序进入不正确的状态。

6.2.2 解决办法

给每一个 sp 增加 2000H 的偏移量，使其远离主程序且指向一片空白的内存区域。

将长度错误的数据段改为正确的长度。

附录 A 参考文献

1. http://www.nasm.us/doc/
   for nasm
2. https://en.wikibooks.org/wiki/X86_Assembly
   for 32bit x86
3. https://en.wikipedia.org/wiki/INT_16H 查阅 wiki 学习 INT 16 中断更详细的用法。
4. 《x86 汇编语言 ++ 从实模式到保护模式完整版》学习了 ORG 伪指令的用处，物理磁盘和虚拟软盘等的工作原理。

附录 B 辅助代码

代码 8: 运行本项目的方法

```
$ sh run.sh
```

代码 9: run.sh 文件内容

```
set -x
bochs -q
```

代码 10: bochs 虚拟机的配置文件

```
# memory
megs: 32

# filename of ROM images
romimage: file=/usr/share/bochs/BIOS-bochs-latest
vgaromimage: file=/usr/share/vgabios/vgabios.bin

floppya: 1_44=boot.img, status=inserted
# floppyb: 1_44=stone.asm, status=inserted

boot: a

log: bochsout.log

mouse: enabled=0

keyboard_mapping: enabled=1, map=/usr/share/bochs/keymaps/x11-pc-us.map

# ips instruction per second
# cpu: count=1, ips=4294967295
```

代码 11: DEBUG 过程中捕捉到的错误信息。栈溢出意外修改到代码段内存

```
; 注意其中的 Caught write watch point at <address>

(0) [0x000000000000a481] 0000:a481 (unk. ctxt): int 0x16                ; cd16
<bochs:32>
(0) Caught write watch point at 0x000000000000a4b0
Next at t=38737314
(0) [0x00000000000f05f1] f000:05f1 (unk. ctxt): push ds                  ; 1e

(0) [0x000000000000a482] 0000:a482 (unk. ctxt): mov ah, 0x01             ; b401
<bochs:29> n
Next at t=32029212
(0) [0x000000000000a484] 0000:a484 (unk. ctxt): int 0x16                 ; cd16
<bochs:30> n
<strong>(0) Caught write watch point at 0x000000000000a4b2</strong>
Next at t=32029254
(0) [0x00000000000f05ee] f000:05ee (unk. ctxt): mov bp, sp               ; 89e5
<bochs:31>

watch 0xa4b3
```