16 级计科 7 班: 操作系统原理实验 #1

Due on Monday, March 12, 2018

凌应标 周一 *9-10* 节

颜彬

16337269

Content

Page

1 实验目的 **3**

2 实验要求 **3**
2.1 搭建和应用实验环境 3
2.2 接管裸机控制权 3

3 实验方案 **3**
3.1 基础原理 3
3.2 实验环境与工具 4
3.3 程序流程 4
3.4 程序模块及相关说明 4

4 实验过程 **7**
4.1 用 Dos 制作引导盘 7
4.2 扇区填充个人信息 7
4.3 接管裸机的控制权 8

5 实验结果 **8**
5.1 代码编译（可选） 8
5.2 运行代码 8
5.3 结果展示 9

6 实验总结 **9**
6.1 心得体会 9
6.2 遇到的 BUG 11

A 参考文献 **11**

B 辅助代码 **11**

1 实验目的

掌握操作系统的相关概念，理解计算机引导的全部过程。利用工具制作正确引导盘并完成特定任务。

理解虚拟机的运行方式，了解虚拟机与真实机器的不同之处。熟悉配置、运行虚拟机的基本方法。掌握在虚拟机下运行程序、程序查错和运行 dos 镜像。

熟悉相关工具的使用。熟悉 32 位 x86 汇编器和二进制文件操作软件的使用。熟悉虚拟镜像的制作方法。掌握基本的 32 位 x86 汇编的编写。掌握汇编程序的编译、运行、调错。掌握硬件级的调试技巧。

2 实验要求

2.1 搭建和应用实验环境

虚拟机安装, 生成一个基本配置的虚拟机 XXXPC 和多个 1.44MB 容量的虚拟软盘, 将其中一个虚拟软盘用 DOS 格式化为 DOS 引导盘, 用 WinHex 工具将其中一个虚拟软盘的首扇区填满你的个人信息。

2.2 接管裸机控制权

设计 ibmpc 的一个引导扇区程序，程序功能从屏幕左边某行位置 45 度角斜向下射出，保持一个可观察的适当速度直线运动，在碰到屏幕边后产生反射，改变方向运动，如此类推，不断运动。在此基础上，增加你的个性拓展，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以特别的方式显示你的学号姓名等个人信息。将这个程序的机器吗放到第三张虚拟软盘的首扇区，并用此软盘引导你的 pc，直到成功。

3 实验方案

3.1 基础原理

实验环境是不带有操作系统的裸机。为了做到接管逻辑控制权和运行程序的目的，我们需要写出自己的引导程序（制作一个引导扇区）。开机后，计算机会首先作自检。若选择从软盘启动，bios 会检查软盘的 0 面 0 磁道 1 扇区, 如果发现它的最后一个字节是 0xAA55，则 bios 会把其视作一个引导扇区，尝试执行其中的程序。

引导扇区通常会将操作系统加载进内存中，将控制权完全交给操作系统，由操作系统完成随后的各种调配。在本实验中，引导扇区内包含了若干代码，用于在屏幕中心高亮显示个人信息，以及完成简单的动画。

即便没有操作系统，本次实验依然可以在终端输出相应的信息，甚至是绘制动画。0xB8000-0xBFFFF 的内存空间是显存地址, 共有 32KB。向这个地址写入数据可以打印到屏幕上。对于从 0xB800 开始的每个字 $16bits$，其高位将解释为输出内容的样式信息，例如前景色、背景色等；其低位将解释为输出内容的 Ascii 码。

通过反复往显存地址中读写数据，本项目最终完成了附带个人资料和动画的引导程序。

3.2 实验环境与工具

3.2.1 实验环境

- 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。
- 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

关于工具 dd 和 xxd 的使用，请见附录B代码13

3.3 程序流程

图1展示了程序 stone.asm 的流程图。

其中” 设定延时” 步骤将初始化一个循环。该循环不断检测某个变量 count 是否已经达到特定的值。若未达到，则将 count 加 1，继续执行下一个循环。若 count 达到了预设的值，则跳出循环，继续执行下面的语句（相当于执行一个 callback function）。当 callback 的语句执行完毕后，末尾的 jmp 指令将把程序再次带入延时循环。

3.4 程序模块及相关说明

3.4.1 实现引导

如代码1所示，在程序的末尾加上该代码片段。

其中 $ 代表当前行的地址。$$ 代表当前扇区的首地址。所以 510-($-$$) 代表当前行距离扇区末尾的长度再减二字节。减二字节是为了在扇区的最后预留空间给 0xAA55, 以使得系统识别该盘为引导盘。

3.4.2 延时循环

代码2展示了延时循环的相关代码。

在延时循环中，代码会不断地将 50000 * 580 这一值减一。当这一值减为 0 时，执行后面的 main: 标签部分, 达到延时的效果。值得注意的是，50000 * 580 这一值较大，无法直接存入一个 32 位变量中，故程序采用 word[count] 和 word[dcount] 两个变量的乘法表示这一整数。

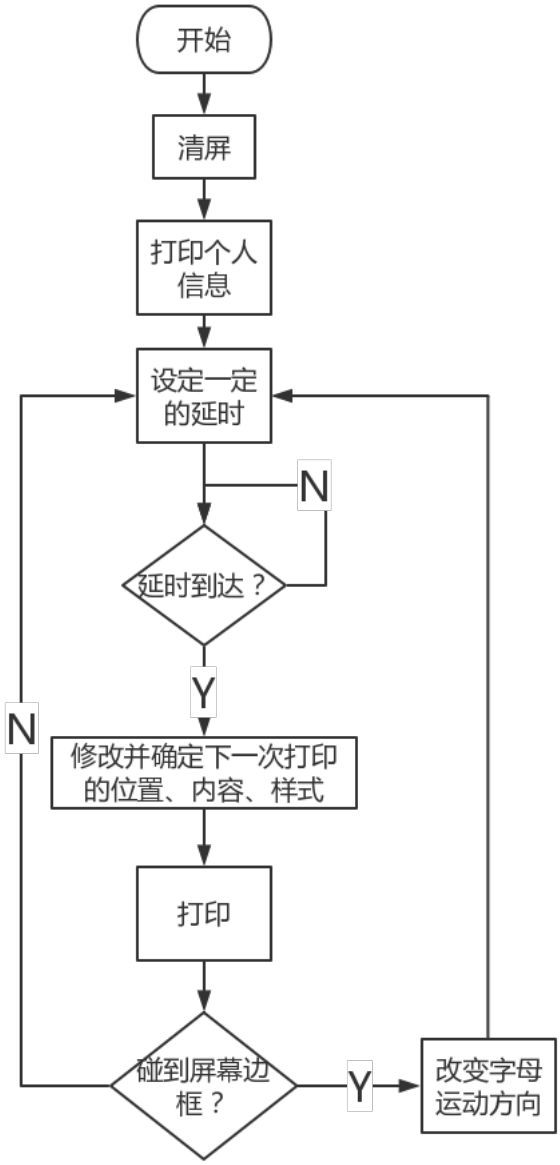

图 1: stone.asm 流程图.

代码 1: 使软盘成为引导盘的关键代码

```
    ; ...
2   times 510-($-$$) db 0
    dw 0xaa55
```

代码 2: 延时循环

```
loop1:
    dec word[count]
    jnz loop1
    mov word[count],delay
    dec word[dcount]
    jnz loop1
    mov word[count],delay
    mov word[dcount],ddelay
main:
    ; code below ...
```

代码 3: 清屏模块

```
ScreenLength equ 80 * 25 * 2
; ...
clearTheScreen:
    mov cx, ScreenLength
    mov bx, 0
clearScreenLoop:
    loop clearWorker
    ret
clearWorker:
    mov word[gs:bx], 0
    inc bx
    jmp clearScreenLoop
```

3.4.3 清屏模块

代码3展示了清屏模块的主要内容。

在清屏模块中，程序首先计算出屏幕大小 ScreenLength（单位字节）。随后进入一个循环，在循环内部反复将 0xB800 随后的 ScreenLength 长度的内存区域置为 0, 达到清屏效果。

代码中，gs 指向显存段地址，即 0xB800.cx 为计数器，在 loop 中逐次-1.

3.4.4 修改显存区

代码4展示了用于修改显存的代码片段。

显示模块的重点在于计算数据的显示位置，以及将数据写入正确的内存地址中

由于屏幕为 25 行 80 列，所以对于 x 行 y 列的数据而言，其索引应为 80x + y. 由于显存采用字长作为显示单元，故该数据应该写入的内存地址为” 显存段首地址:2*(80x + y)”。在整个程序中，gs 已被初始化为显存段地址 0xB800, 故 [gs:bx] 将指向写入的内存地址。

代码4省略了部分细枝末节的部分。代码假定 ax 中存储了打印的有关信息，其中高位 ah 存储了打印的

代码 4: show 模块

```
show:
    ; ...
    xor ax,ax ; set ax to 0
    mov ax,word[x]
    mov bx,80
    mul bx
    add ax,word[y] ; ax = 80 * x + y
    mov bx,2
    mul bx
    mov bx,ax ; bx = 2(80 * x + y)
    ; ...
    mov [gs:bx],ax
    ret
```

代码 5: 创建虚拟软盘

```
$ bximage
    // ... ignore outputs

$ bochs
...format B:
```

样式，例如前景色和背景色等；低位 al 存储了打印的字符对应的 ASCII 码。

4 实验过程

4.1 用 Dos 制作引导盘

首先需要创建一个虚拟软盘。如代码5所示，在 linux 下采用 bximage 命令，产生一个 1.44Mb, 命名为 a.img 的虚拟镜像。本实验采用 bochs 虚拟机，故首先需要在 bochs 官网下下载 FreeDos 镜像。镜像下载完毕后，解压，将其中的 a.img 文件重命名为 freedos.img。

bochs 默认采用.bochsrc 文件为配置文件。按照代码6的方式配置 bochs, 使其以 freedos.img 为 A 盘，以空镜像 a.img 为 B 盘，并设置 A 盘为启动盘。

随后，如代码5后半部分所示，运行 bochs，进入 dos 系统，格式化 B 盘。此时 B 盘即成为引导盘。

4.2 扇区填充个人信息

本问题采用更灵活的方式实现。

如代码7所示，利用 nasm 的语法特性，产生数个个人信息的字符串，用于填满软盘空间。随后，如代码8所示，直接将编译后的所有二进制信息写入 fill.img 中。此时 fill.img 内即填满了所有的个人信息。

代码 6: .bochsrc 文件部分节选

```
...
floppya: 1_44=freedos.img, status=inserted
floppyb: 1_44=a.img, status=inserted

boot: a
...
```

代码 7: 来自 fill.asm 文件内容

```
times 32 db "16337269 yanbin "
```

4.3 接管裸机的控制权

如3.4节和3.3所示，按照该思路实现 stone.asm 文件，并采用5.1节所示的方法编译，5.2节所示的方法运行代码。结果见5.3节。

5 实验结果

5.1 代码编译（可选）

在项目文件中已经包括了产生的所有代码。所以这一步可以跳过
代码 9展示了将汇编最终写入镜像文件的方法。它将首先使用 nasm 将.asm 文件编译为.bin 文件。随后使用 dd 将.bin 的内容写入.img 中。

代码 9: 将汇编转化为二进制并写入镜像

```
    $ bximage # 先用该命令产生空白的stone.img文件
    $ sh build.sh
```

代码9的内容请见附录B的代码13

5.2 运行代码

按代码10的方式运行脚本，即可运行 stone.asm 程序。在这一步中，脚本会调用 bochs 运行完成好的镜像文件。bochs 将自动加载相同目录内的 .bochsrc 文件，以此为配置运行虚拟机。

代码 10: 运行方法

```
$ sh run.sh
```

代码10的内容见附录B代码14

代码 8: 编译 fill.asm 并将内容写入 fill.img

```
$ nasm fill.asm -o fill.bin
$ dd if=fill.bin of=fill.img
```

代码 11: Dos 下制作的引导盘 a.img 的部分内容

```
00000000: eb3c 9046 7265 6544 4f53 0000 0201 0100  .<.FreeDOS......
00000010: 02e0 0040 0bf0 0900 1200 0200 0000 0000  ...@............
00000020: 0000 0000 0100 2900 0000 0020 2020 2020  ......)....
00000030: 2020 2020 2020 4641 5431 3220 2020 0e1f        FAT12   ..
00000040: be5b 7cac 22c0 740b 56b4 0ebb 0700 cd10  .[|.".t.V.......
00000040: be5b 7cac 22c0 740b 56b4 0ebb 0700 cd10  .[|.".t.V.......
00000050: 5eeb f032 e4cd 16cd 19eb fe54 6869 7320  ^..2.......This
00000060: 6973 206e 6f74 2061 2062 6f6f 7461 626c  is not a bootabl
00000070: 6520 6469 736b 2e20 2050 6c65 6173 6520  e disk.  Please
00000080: 696e 7365 7274 2061 2062 6f6f 7461 626c  insert a bootabl
00000090: 6520 666c 6f70 7079 2061 6e64 0d0a 7072  e floppy and..pr
000000a0: 6573 7320 616e 7920 6b65 7920 746f 2074  ess any key to t
000000b0: 7279 2061 6761 696e 202e 2e2e 200d 0a00  ry again ... ...
......
000001f0: 0000 0000 0000 0000 0000 0000 0000 55aa  ..............U.
```

代码 12: 节选自 fill.img

```
00000000: 3136 3333 3732 3639 2079 616e 6269 6e20  16337269 yanbin
00000010: 3136 3333 3732 3639 2079 616e 6269 6e20  16337269 yanbin
......
000001e0: 3136 3333 3732 3639 2079 616e 6269 6e20  16337269 yanbin
000001f0: 3136 3333 3732 3639 2079 616e 6269 6e20  16337269 yanbin
```

5.3 结果展示

5.3.1 Dos 制作的启动盘

代码11列出了由 dos 格式化后得到的启动盘的具体信息。

5.3.2 写满个人信息的虚拟软盘

代码12展示了文件 fill.img 的部分情况。

5.3.3 引导程序运行状况

图片2展示了在 bochs 下运行 stone.asm 机器码的实际图片。其中图中心的个人信息会不断闪烁高亮。

6 实验总结

6.1 心得体会

这次实验，从配置环境开始，就是对我自己的一大挑战。

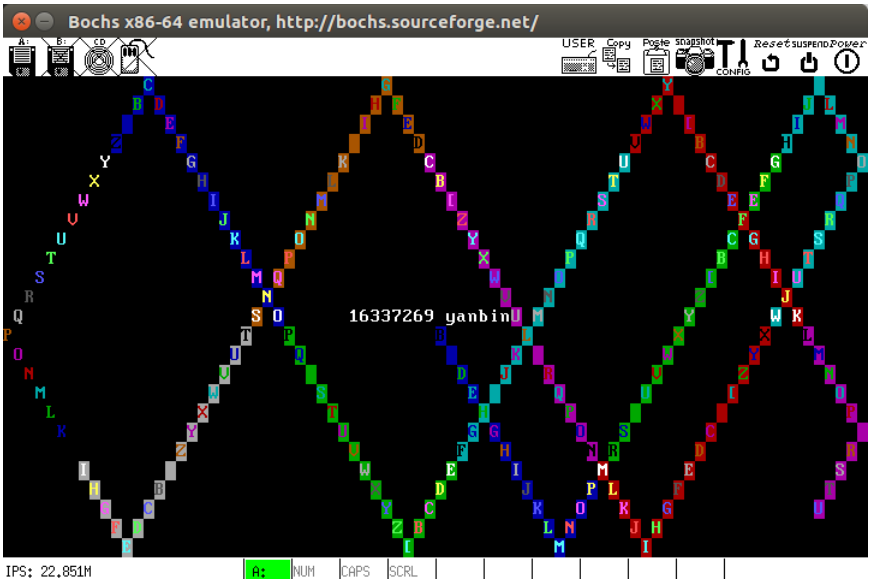

图 2: stone.asm 在 bochs 下的运行截图

在阅读了《ORANGE' S: 一个操作系统的实现. 于渊》的前两章后，我发现它的内容与本次实验比较契合。于是我就萌生了按照这本书的方式配置和实现本项目的想法。由于我常年用 Linux 系统，所以从最开始，我便一直考虑着在 Linux 下完成操作系统的所有项目。这刚好又于上书是契合的。上书也主要采用 Linux 系统作为讲解。

选择了 Linux，一方面意味着众多极其强大的开源软件，另一方面也意味着，我将走和许多同学不同的道路，用几乎完全不同的工具链。毕竟老师给出的工具包的内容几乎是 Windos 下的开发工具。我其实并不担心工具的问题，因为使用 Linux 的经验告诉我，Linux 的组件往往比 Windows 更丰富，配置也更简单，摸索出属于自己的工具链并不是一件难事。我最担心的问题是，身边的同学如果大部分使用 Windows 来开发，那么我在 Linux 下遇到问题将很难找到讨论的伙伴。所幸，我的朋友王永锋同学也选择 Linux 下开发，于是这敲定了我所使用的环境，并让我配置出了自己较为舒适的工具链。nasm 编译，bochs 作为虚拟机运行程序和调试程序。dd, bximage, xxd 命令予以协助。

令人感到不适的是，网上关于 nasm 和 bochs 的相关讨论非常少。在 stackoverflow 上几乎搜不到我遇到的关于 nasm 的讨论。但好在 nasm 的官方文档写得比较用心，而且我个人的英语也不差，于是把文档的前几章大概看完了。看完的同时我也体会到 nasm 比别的汇编器更强大的地方。可以说官方文档几乎成为了我学习、查询 nasm 语法的唯一方法。

对程序的 debug 也是这个项目的一大挑战。刚好 bochs 自带调试功能，这给我带来了很大的便利。bochs 在 google 上也几乎没有第三方的 tutorial，我也只好把 bochs 的 debug 文档通读了一遍。bochs 几乎具有我所期望的 debugger 的所有需求，唯一让我失望的是它的反汇编功能（个人认为）还不够强大。如果我想要把汇编文档的语句对应到 bochs 调试时执行的语句，我需要指定语句的内存地址。然而在实际调试中，确定内存地址是一件很繁琐的事情。但在其他方面 bochs 都足够让我调试程序了。与上古程序员相比，我们的工具其实已经很丰富了。

本次实验，我的收获十分巨大。在上这门课之前，我对操作系统几乎一无所知。对我而言操作系统就如一个黑箱，我仅仅知道它是一个程序，它负责调配资源，并把极其复杂的硬件底层信息隔离在程序员的

视线之外。在经过这个实验之后，最起码我已经明白了操作系统最初是如何进入到内存中的。

这个实验给我带来的成就感最大的地方在于，我独立地制作出了引导盘，并在上面运行了我的 DIY 程序。在实验开始前，我十分惧怕与硬件打交道，害怕硬件中繁复的细节。但在我的最简单的引导正确输出了 “helloworld” 之后，我意识到，硬件也并没有我想的那么复杂。硬件有它自己的规定，按照硬件的规定实现代码，我们依旧可以完全地控制硬件，让它为我们所用。

凌老师曾有过一个比喻，这个实验 “就像一个种子，它落到土里会生根发芽”。虽然这个引导程序几乎没有干十分特别的事，但它是独立运行在裸机上的。这就意味着，这个引导程序将成为我们实现操作系统的根基。以后实验的所有代码会直接依赖于这一段程序。。
希望我能在以后的实验中，对操作系统的各种概念理解得更加深刻，逐步完成好属于我自己的操作系统。

6.2 遇到的 BUG

6.2.1 MUL 指令

16 位和 32 位的 MUL 指令会将高位的进位置于 dx/edx 寄存器中。我一开始并没有留意到这个情况只是发现 dx 寄存器会被莫名其妙地清零。现象可复现。

发现这个 bug 并解决的方法是使用 bochs 调试。我先找到 “因为 dx 清零而出错” 的地方，在这个地方设置断点，单步执行并输出所有的寄存器的值。最终发现在 MUL 指令的前后 dx 的值发生了变化。通过参考 wikibook 的解释，我理解了 bug 的原因，并修改了代码

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

附录 B 辅助代码

代码 13: build.sh 文件内容

```
set -x
nasm stone.asm -o stone.bin
dd if=stone.bin of=stone.img bs=512 count=1 conv=notrunc
```

代码 14: run.sh 文件内容

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

代码 15: .bochsrc 文件，作为 bochs 的配置文件以运行 stone.asm

```
# 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=stone.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
```