

摘 要

操作系统设计与实现实验的实验目的是：通过设计动手设计一个操作系统，增加对于操作系统的直观认识，深刻了解进程角度、输入输出、系统时钟、保护模式等有关操作系统方面的知识，并且熟悉有关虚拟机的相关操作。

实验设计主要遵循：操作系统的相关知识、NASM 汇编语言、C 语言。

实验内容主要包括：搭建包含 Linux 虚拟机和 bochs 仿真工具实验环境，并完成《ORANGE'S：一个操作系统的实现》中前 7 章代码的实现。

实验结论为：代码可以调试成功。

关键词：操作系统；汇编语言；C 语言；虚拟机；bochs

目 录

1 马上动手写一个最小的“操作系统”	1
1.1 十分钟完成操作系统	1
1.1.1 关键代码	1
1.1.2 主要代码结构	1
1.1.3 疑难理解	2
1.1.4 调试过程	2
1.1.5 实验分析	4
2 搭建你的工作环境	6
2.1 VMware Workstation	6
2.2 Ubuntu	6
2.3 Bochs	6
2.4 NASM、GCC 和 GNU MAKE	7
3 保护模式	9
3.1 进入保护模式	9
3.1.1 关键代码	9
3.1.2 主要代码结构	11
3.1.3 调试过程	12
3.1.4 实验分析	13
3.2 保护模式进阶——从保护模式返回实模式	14
3.2.1 关键代码	14
3.2.2 主要代码结构	23
3.2.3 调试过程	24
3.2.4 实验分析	25
3.3 局部描述符表 LDT	27
3.3.1 关键代码	27
3.3.2 主要代码结构	29
3.3.3 调试过程	30
3.3.4 实验分析	31
4 让操作系统走进保护模式	32
4.1 DOS 可以识别的引导盘	32
4.1.1 关键代码	32

4.1.2 主要代码结构	33
4.1.3 调试过程	34
4.1.4 实验分析	34
4.2 实验 2 一个最简单的 Loader	36
4.2.1 关键代码	36
4.2.2 主要代码结构	40
4.2.3 调试过程	40
4.2.4 实验分析	42
4.3 实验 3 向 Loader 交出控制权	43
4.3.1 关键代码	43
4.3.2 主要代码结构	46
4.3.3 调试过程	47
4.3.4 实验分析	47
5 内核雏形	49
5.1 在 Linux 下用汇编写 Hello World	49
5.1.1 关键代码	49
5.1.2 主要代码结构	50
5.1.3 调试过程	50
5.1.4 实验分析	50
5.2 再进一步，汇编和 C 同步使用	51
5.2.1 关键代码	51
5.2.2 主要代码结构	53
5.2.3 调试过程	53
5.2.4 实验分析	54
5.3 从 Loader 到内核	55
5.3.1 关键代码	55
5.3.2 主要代码结构	61
5.3.3 调试过程	62
5.3.4 实验分析	63
6 进程	64
6.1 最简单的进程	64
6.1.1 关键代码	64
6.1.2 代码主要结构	66

6.1.3 调试过程	66
6.1.4 实验分析	69
6.2 丰富中断处理程序	71
6.2.1 关键代码	71
6.2.2 代码主要结构	72
6.2.3 调试过程	73
6.2.4 实验分析	73
6.3 中断重入	75
6.3.1 关键代码	75
6.3.2 代码主要结构	76
6.3.3 调试过程	77
6.3.4 实验分析	77
6.4 功能改进	79
7 输入/输出系统	84
7.1 从中断开始——键盘初体验	84
7.1.1 关键代码	84
7.1.2 代码主要结构	84
7.1.3 调试过程	85
7.1.4 实验分析	85
7.2 从缓冲区读取信息并打印读取的值	86
7.2.1 关键代码	86
7.2.2 代码主要结构	86
7.2.3 调试过程	86
7.2.4 实验分析	87
7.3 扫描码解析数组、键盘输入缓冲区、与使用新加的任务处理键盘操作	88
7.3.1 关键代码	88
7.3.2 代码主要结构	90
7.3.3 调试过程	90
7.3.4 实验分析	91
7.4 解析扫描码——让字符显示出来	92
7.4.1 关键代码	92
7.4.2 代码主要结构	93
7.4.3 调试过程	93

7.4.4 实验分析	94
7.5 功能改进	95
8 实验总结	103
8.1 实验中经常遇到的错误	103
8.1.1 bochs 配置文件 bochsrc 有关的问题	103
8.1.2 出现挂载点/mnt/floppy 不存在的提示	104
8.1.3 ld 指令出现 incompatible with i386:x86-64 output	104
8.1.4 gcc 指令相关问题	104
8.1.5 调试过程中出现乱码、红色错误、输出“error”等问题	105
8.2 实验总结与心得	108
8.2.1 实验总结	108
8.2.2 实验心得	116
9 参考文献	118
10 教师评语评分	119

1 马上动手写一个最小的“操作系统”

1.1 十分钟完成操作系统

1.1.1 关键代码

代码段 1.1 boot.asm

```
org 07c00h          ; 告诉编译器程序加载到 7c00 处

mov ax, cs
mov ds, ax
mov es, ax

call DispStr        ; 调用显示字符串例程

jmp $               ; 无限循环

DispStr:
    mov ax, BootMessage
    mov bp, ax      ; ES:BP = 串地址
    mov cx, 16      ; CX = 串长度
    mov ax, 01301h   ; AH = 13, AL = 01h
    mov bx, 000ch    ; 页号为 0(BH = 0) 黑底红字(BL = 0Ch,高亮)
    mov dl, 0
    int 10h         ; 10h 号中断
    ret

BootMessage:        db "Hello, OS world!"
times 510-($-$$)    db 0 ; 填充剩下的空间, 使生成的二进制代码恰好为 512 字节
dw 0xaa55           ; 结束标志
```

1.1.2 主要代码结构

- 告诉编译器程序加载到 07c00 处
- 使 ds 和 es 两个段寄存器指向与 cs 相同的段
- 调用 DispStr 函数显示字符串
- 无限循环

DispStr 函数:

- 设置 ES:BP = 串地址
- 设置 CX = 串长度
- 设置 AH = 13, AL = 01h
- 设置页号为 0(BH = 0) 黑底红字(BL = 0Ch,高亮)
- 10h 号中断

1.1.3 疑难理解

- 在 nasm 中，任何不被 [] 括起来的标签名或变量均为地址，例如“mov ax, BootMessage”中，BootMessage 为字符串的首地址。
- \$表示当前行被汇编后的地址，\$\$表示一个节的开始处被汇编后的地址，因此“\$-\$”表示当前行的相对地址。则“times 510-(\$-\$) db 0”表示把 0 这个字节重复 510-(\$-\$) 遍，使程序占满 510 个字节，最后 dw 0xaa55 加上结束标志后正好为 512 个字节。
- **代码功能：**代码通过 mov 指令使 ds 和 es 两个段寄存器指向 cs 相同的段之后，把字符串首地址移到 bp 中，然后设置字符串的颜色等，最后“int 10h”指令通过调用 10 号中断显示字符串。

1.1.4 调试过程

想要进行操作系统的调试，首先必须创建一个软盘映像，为了达成这个目标，必须使用 bimage 软件，在控制台中输入 bimage 即可打开相应的界面。

在打开 bimage 后，依次输入 1->fd->回车->回车，即可生成一个名为 a.img 的软盘映像。

创建完软盘映像后，需要编译源代码，输入代码：

nasm boot.asm -o boot.bin

即可完成编译。

完成编译之后，需要使用软盘绝对扇区读写工具将这个文件写到一张空白软盘的第一个扇区，输入代码：

dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc

即可完成读入。

一切准备就绪，紧接着需要编写配置文件，即告诉 Bochs 虚拟机的具体信息，

如内存多大、硬盘映像和软盘映像都是那些文件的内容，以下为 Bochs 配置文件 bochsrc 的示例：

代码段 1.2 bochsrc

```
#####  
# Configuration file for Bochs  
#####  
  
# how much memory the emulated machine will have  
megs: 32  
  
# filename of ROM images  
romimage: file=/usr/share/bochs/BIOS-bochs-latest  
# vgaromimage: /usr/share/vgabios/vgabios.bin  
vgaromimage: file=/usr/share/vgabios/vgabios.bin  
  
# what disk images will be used  
floppya: 1_44=a.img, status=inserted  
  
# choose the boot disk.  
boot: floppy  
  
# where do we send log messages?  
# log: bochsout.txt  
  
# disable the mouse  
mouse: enabled=0  
  
# enable key mapping, using US layout as default.  
# keyboard_mapping: enabled=1, map=/usr/share/bochs/keymaps/x11-pc-us  
.map
```



```
keyboard: keymap=/usr/share/bochs/keymaps/x11-pc-us.map
```

需要注意的是，书本配套代码中的配置文件是错误的，需要在 `vgaromimage` 和 `keyboard` 处进行修改，即上述代码中的标红部分。改正后的代码均在错误代码的下一行。

完成上述三个步骤之后，一切准备就绪，可以正式调试，输入命令：

bochs -f bochs.rc

即可开始调试过程。

这时，需要再输入 `c`（代表开始调试），再按下回车键，即会进入 `bochs` 页面，这时，在屏幕的左上角会出现一行红色的“Hello, OS world!”，即代表调试成功，调试结果图 1.1 所示。

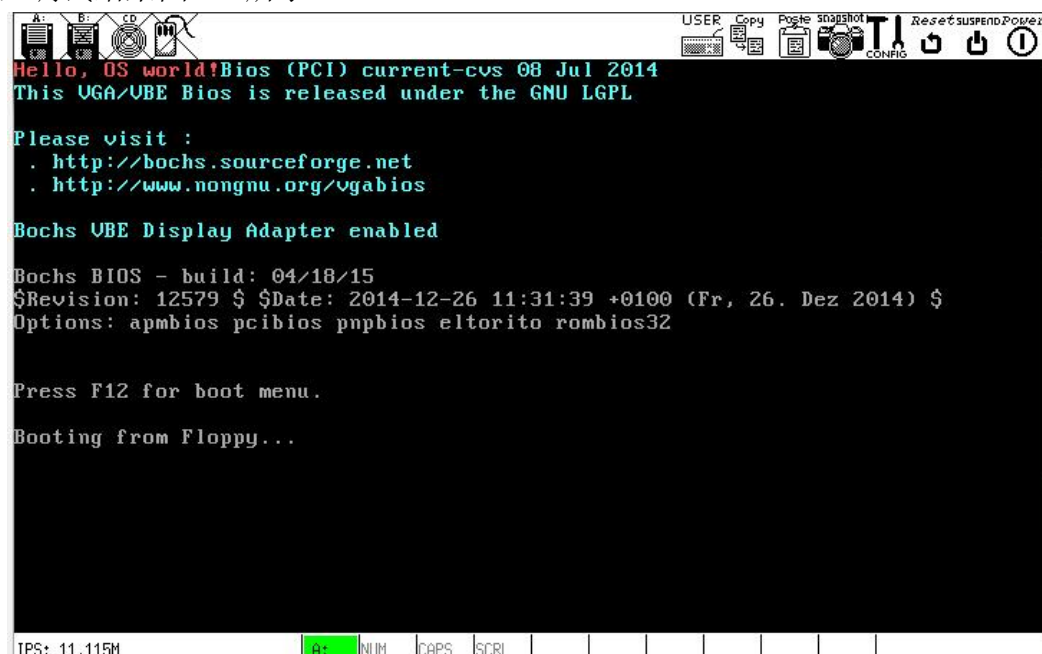


图 1.1 调试结果

1.1.5 实验分析

在本实验中，笔者使用汇编语言编写了一个最简单的操作系统，并且使用 Bochs 软件进行仿真。

事实上，笔者编写的程序并不能成为“操作系统”，而只能称为一个最简单的引导扇区（Boot Sector），但无论如何，它是脱离操作系统、在裸机上运行的。

但计算机电源打开时，它首先会加电自检（POST），然后寻找启动盘，如果是选择从软盘启动，计算机就会检查软盘的 0 面 0 磁道 1 扇区，如果发现它以

0xAA55 结束，那么 BIOS 就会认为它是一个引导扇区。当然，一个正确的引导扇区除了以 0xAA55 结束之外，还应该包含一段少于 512 字节的执行码。

而一旦 BIOS 发现了引导扇区，就会将这 512 字节的内容装载到内存地址 0000:7c00 处，然后跳转到 0000:7c00 处，将控制权彻底交给这段引导代码。至此为止，计算机不再有 BIOS 中固有的程序来控制，而是变为由操作系统的一部分来控制。

所以本实验的关键在于引导扇区的创建。即代码的最后部分：

代码段 1.3 boot.asm

```
times    510-($-$$)    db    0    ; 填充剩下的空间，使生成的二进制代码恰好为 512 字节
dw    0xaa55            ; 结束标志
```

2 搭建你的工作环境

2.1 VMware Workstation

VMware Workstation 允许操作系统(OS)和应用程序在一台虚拟机内部运行。虚拟机是独立运行主机操作系统的离散环境。在 VMware Workstation 中,你可以在一个窗口中加载一台虚拟机,它可以运行自己的操作系统和应用程序。

在《ORANGE'S: 一个操作系统的实现》一书的实验中,笔者选择在 Ubuntu 环境下运行 bochs、nasm、gcc 等工具,因此需要下载 VMware 运行 Ubuntu 虚拟机。

2.2 Ubuntu

Ubuntu 是一个基于 Debian 的以桌面应用为主的 Linux 操作系统。Ubuntu 的目标在于为一般用户提供一个最新同时又相当稳定,主要以自由软件建构而成的操作系统。

官网下载相应的镜像文件 ubuntu-20.04.2.0-desktop-amd64.iso,在 VMware 中配置生成相应 Ubuntu 64 位虚拟机。

2.3 Bochs

为了能够对编写的源代码进行编译和仿真,还必须下载相应的程序。

对于仿真程序,《ORANGE'S: 一个操作系统的实现》一书中选择使用 Bochs 进行仿真。

Bochs 是一个 x86 硬件平台的开源模拟器。它可以模拟各种硬件的配置。Bochs 模拟的是整个 PC 平台,包括 I/O 设备、内存和 BIOS。更为有趣的是,甚至可以不使用 PC 硬件来运行 Bochs。事实上,它可以在任何编译运行 Bochs 的平台上模拟 x86 硬件。通过改变配置,可以指定使用的 CPU(386、486 或者 586),以及内存大小等。

安装过程如下:

第一步: 在终端执行以下命令,安装几个包:

```
sudo apt-get install build-essential nasm
```

```
sudo apt-get install libx11-dev
```

```
sudo apt-get install xorg-dev
```

sudo apt-get install libgtk2.0-dev

sudo apt-get install bison

第二步：解压下载的 bochs 安装包：**tar zxvf bochs-2.6.9.tar.gz**

第三步：进入解压后的目录：**cd bochs-2.6.9**

第四步：执行：**./configure --enable-debugger --enable-disasm**

第五步：进行编译：**sudo make**

第六步：安装，输入命令：**sudo make install**

安装完成后，在终端输入命令：**bochs**，显示如图 2.1 所示。

```
=====
00000000000i[      ] BXSHARE not set. using compile time default '/usr/local/sha
re/bochs'
-----
Bochs Configuration: Main Menu
-----

This is the Bochs Configuration Interface, where you can describe the
machine that you want to simulate.  Bochs has already searched for a
configuration file (typically called bochsrc.txt) and loaded it if it
could be found.  When you are satisfied with the configuration, go
ahead and start the simulation.

You can also start bochs with the -q option to skip these menus.

1. Restore factory default configuration
2. Read options from...
3. Edit options
4. Save options to...
5. Restore the Bochs state from...
6. Begin simulation
7. Quit now

Please choose one: [2] |
```

图 2.1 bochs 安装成功后的显示

2.4 NASM、GCC 和 GNU MAKE

在《ORANGE’S：一个操作系统的实现》一书中，所编写的操作系统是由汇编语言和 C 语言共同完成的，为了编译汇编语言，必须安装 NASM 程序，为了编译 C 语言，必须安装 NASM 程序，同时，还必须安装 GNU Make，用于自动

化编译和链接。

在 Ubuntu 中，已经预安装了 GCC 和 NASM 这两个程序，其余部分包，已经在 2.3 节步骤中的第一步安装完成。

3 保护模式

3.1 进入保护模式

3.1.1 关键代码

代码段 3.1 pmtest1.asm

```
...  
[SECTION .s16]  
[BITS 16]  
LABEL_BEGIN:  
    mov ax, cs  
    mov ds, ax  
    mov es, ax  
    mov ss, ax  
    mov sp, 0100h  
  
    ; 初始化 32 位代码段描述符  
    xor eax, eax  
    mov ax, cs  
    shl eax, 4  
    add eax, LABEL_SEG_CODE32  
    mov word [LABEL_DESC_CODE32 + 2], ax  
    shr eax, 16  
    mov byte [LABEL_DESC_CODE32 + 4], al  
    mov byte [LABEL_DESC_CODE32 + 7], ah  
  
    ; 为加载 GDTR 作准备  
    xor eax, eax  
    mov ax, ds  
    shl eax, 4  
    add eax, LABEL_GDT    ; eax <- gdt 基地址
```

```

mov dword [GdtPtr + 2], eax ; [GdtPtr + 2] <- gdt 基地址

; 加载 GDTR
lgdt [GdtPtr]

; 关中断
cli

; 打开地址线 A20
in al, 92h
or al, 00000010b
out 92h, al

; 准备切换到保护模式
mov eax, cr0
or eax, 1
mov cr0, eax

; 真正进入保护模式
jmp dword SelectorCode32:0 ; 执行这一句会把 SelectorCode32 装入 cs,
                                ; 并跳转到 Code32Selector:0 处
; END of [SECTION .s16]

[SECTION .s32]; 32 位代码段. 由实模式跳入.
[BITS 32]

LABEL_SEG_CODE32:

mov ax, SelectorVideo
mov gs, ax ; 视频段选择子(目的)

```

```

mov edi, (80 * 11 + 79) * 2    ; 屏幕第 11 行, 第 79 列。

mov ah, 0Ch                    ; 0000: 黑底    1100: 红字

mov al, 'P'

mov [gs:edi], ax

; 到此停止

jmp $

```

```

SegCode32Len equ $ - LABEL_SEG_CODE32

```

```

; END of [SECTION .s32]

```

3.1.2 主要代码结构

3.1.2.1 代码段[SECTION .s16]: 从实模式跳到保护模式

- 初始化 32 位代码段描述符:将[SECTION .S32]段的物理地址分三部分赋给 0 描述符 DESC_CODE32
- 为加载 GDT 做准备:将 GDT 的物理地址填充到 GdtPtr 中
- 加载 GDT:lgdt [GdtPtr] #将 GdtPtr 加载到寄存器 gdtr 中
- 关中断
- 打开 A20 地址线:通过操作端口 92h
- 准备切换到保护模式:把寄存器 cr0 的第 0 位置为 1
- 进入保护模式: Jump

3.1.2.2 代码段[SECTION .s32]: 实模式跳入 32 位代码段并在屏幕上输出

- 将视频段选择子 SelectorVideo 的地址赋给 gs
- 设置输出位置为屏幕 11 行 79 列
- 设置输出为黑底红字
- 往显存中 edi 偏移处 ([gs:edi]地址处) 写入 P
- 进入无限循环

3.1.3 调试过程

在之前的实验过程中，将文件写到了引导扇区运行，这样比较方便，但引导扇区的空间有限，只有 512 个字节，为了使程序可以扩大可以选择先借用其他操作系统的引导扇区，这里《ORANGE'S：一个操作系统的实现》一书选择借用 DOS 操作系统，因此必须将程序编译成 COM 文件，然后用 DOS 来执行它。

为了这么做，第一步需要到 Bochs 官方网站下载一个 FreeDos，解压后将其中的 a.img 复制到工作目录中，并改名为 freedos.img。

紧接着，需要使用 bximage 生成一个软盘映像，取名为 pm.img。再其次，需要修改 bochsrc，与第 1 章里中提到的修改方法相同。创建完软盘映像后，需要编译源代码，输入代码：

```
nasm pmtest1.asm -o pmtest1.bin
```

即可完成编译。

完成编译之后，需要使用软盘绝对扇区读写工具将这个文件写到一张空白软盘的第一个扇区，输入代码：

```
dd if=pmtest1.bin of=a.img bs=512 count=1 conv=notrunc
```

即可完成读入。

接下来，输入 **bochs** 并按下回车键，则启动 Bochs；输入 **c** 并且按下回车键，则开始调试；输入 **B:\pmtest1.com** 并按下回车键，则 pmtest1.com 开始运行，运行结果如图 3.1 所示。



图 3.1 pmtest1.com 运行结果

可以注意到，一个红色的字母“P”出现在了屏幕的右侧的中部，这代表调试成功，程序已经进入了保护模式。

3.1.4 实验分析

在 IA32 下，CPU 有两种工作模式：实模式和保护模式，但打开计算机时，开始时 CPU 是工作在实模式下的，但笔者需要让其进入保护模式，发挥其巨大的寻址能力，并为强大的 32 位系统提供硬件保障。

在本实验中，笔者在 `pm.inc` 文件中定义了全局描述表 GDT，定义了相应的段描述符，并且通过代码由实模式进入了保护模式，最后在保护模式中，通过代码在屏幕右边中央打印了一个红色的“P”，代表实验成功。

3.2 保护模式进阶——从保护模式返回实模式

3.2.1 关键代码

代码段 3.2 pmtest2.asm

```
...  
[SECTION .s16]  
[BITS 16]  
LABEL_BEGIN:  
  
    mov ax, cs  
  
    mov ds, ax  
  
    mov es, ax  
  
    mov ss, ax  
  
    mov sp, 0100h  
  
  
    mov [LABEL_GO_BACK_TO_REAL+3], ax  
    mov [SPValueInRealMode], sp  
  
  
; 初始化 16 位代码段描述符  
  
    mov ax, cs  
  
    movzx    eax, ax  
  
    shl     eax, 4  
  
    add     eax, LABEL_SEG_CODE16  
  
    mov word [LABEL_DESC_CODE16 + 2], ax  
  
    shr     eax, 16  
  
    mov byte [LABEL_DESC_CODE16 + 4], al  
    mov byte [LABEL_DESC_CODE16 + 7], ah  
  
  
; 初始化 32 位代码段描述符  
  
    xor     eax, eax  
  
    mov ax, cs
```

```

shl  eax, 4

add  eax, LABEL_SEG_CODE32

mov  word [LABEL_DESC_CODE32 + 2], ax

shr  eax, 16

mov  byte [LABEL_DESC_CODE32 + 4], al

mov  byte [LABEL_DESC_CODE32 + 7], ah

```

； 初始化数据段描述符

```

xor  eax, eax

mov  ax, ds

shl  eax, 4

add  eax, LABEL_DATA

mov  word [LABEL_DESC_DATA + 2], ax

shr  eax, 16

mov  byte [LABEL_DESC_DATA + 4], al

mov  byte [LABEL_DESC_DATA + 7], ah

```

； 初始化堆栈段描述符

```

xor  eax, eax

mov  ax, ds

shl  eax, 4

add  eax, LABEL_STACK

mov  word [LABEL_DESC_STACK + 2], ax

shr  eax, 16

mov  byte [LABEL_DESC_STACK + 4], al

mov  byte [LABEL_DESC_STACK + 7], ah

```

； 为加载 GDTR 作准备

```

xor  eax, eax

mov  ax, ds

```

```

    shl     eax, 4

    add     eax, LABEL_GDT      ; eax <- gdt 基地址

    mov     dword [GdtPtr + 2], eax ; [GdtPtr + 2] <- gdt 基地址

; 加载 GDTR
    lgdt    [GdtPtr]

; 关中断
    cli

; 打开地址线 A20
    in      al, 92h
    or      al, 00000010b
    out     92h, al

; 准备切换到保护模式
    mov     eax, cr0
    or      eax, 1
    mov     cr0, eax

; 真正进入保护模式
    jmp     dword SelectorCode32:0 ; 执行这一句会把 SelectorCode32 装入 cs, 并跳转到
Code32Selector:0 处

; =====

LABEL_REAL_ENTRY:      ; 从保护模式跳回到实模式就到了这里

    mov     ax, cs
    mov     ds, ax
    mov     es, ax

```

```

mov ss, ax

mov sp, [SPValueInRealMode]

in  al, 92h          ;`.
and al, 1111101b    ; | 关闭 A20 地址线
out 92h, al          ;/

sti                  ; 开中断

mov ax, 4c00h ;`.
int 21h             ;/ 回到 DOS
; END of [SECTION .s16]

```

[SECTION .s32]; 32 位代码段. 由实模式跳入.

[BITS 32]

LABEL_SEG_CODE32:

```

mov ax, SelectorData
mov ds, ax          ; 数据段选择子
mov ax, SelectorTest
mov es, ax          ; 测试段选择子
mov ax, SelectorVideo
mov gs, ax          ; 视频段选择子

mov ax, SelectorStack
mov ss, ax          ; 堆栈段选择子

mov esp, TopOfStack

```

```

; 下面显示一个字符串

mov ah, 0Ch          ; 0000: 黑底    1100: 红字

xor esi, esi

xor edi, edi

mov esi, OffsetPMMMessage ; 源数据偏移

mov edi, (80 * 10 + 0) * 2 ; 目的数据偏移。屏幕第 10 行, 第 0 列。

cld

.1:

    lodsb

    test al, al

    jz .2

    mov [gs:edi], ax

    add edi, 2

    jmp .1

.2: ; 显示完毕

    call DispReturn

    call TestRead

    call TestWrite

    call TestRead

; 到此停止

jmp SelectorCode16:0

; -----

TestRead:

    xor esi, esi

```

```

    mov ecx, 8
.loop:

    mov al, [es:esi]

    call DispAL

    inc esi

    loop .loop

    call DispReturn

    ret

; TestRead 结束-----

; -----

TestWrite:

    pushesi
    pushedi

    xor esi, esi
    xor edi, edi

    mov esi, OffsetStrTest ; 源数据偏移
    cld

.1:
    lodsb

    test al, al

    jz .2

    mov [es:edi], al

    inc edi

    jmp .1

.2:

```



```

    pop edi

    pop esi

    ret

; TestWrite 结束-----

; -----

; 显示 AL 中的数字
; 默认的:
;   数字已经存在 AL 中
;   edi 始终指向要显示的下一个字符的位置
; 被改变的寄存器:
;   ax, edi
; -----

DispAL:

    push ecx
    pushedx

    mov ah, 0Ch          ; 0000: 黑底    1100: 红字
    mov dl, al
    shr al, 4
    mov ecx, 2

.begin:
    and al, 01111b
    cmp al, 9
    ja  .1
    add al, '0'
    jmp .2

.1:

```

```

    sub al, 0Ah

    add al, 'A'

.2:

    mov [gs:edi], ax
    add edi, 2

    mov al, dl
    loop .begin
    add edi, 2

    pop edx
    pop ecx

    ret

; DispAL 结束-----

; -----

DispReturn:

    pusheax
    pushebx
    mov eax, edi
    mov bl, 160
    div bl
    and eax, 0FFh
    inc eax
    mov bl, 160
    mul bl
    mov edi, eax
    pop ebx

```

```

    pop    eax

    ret

; DispReturn 结束-----

SegCode32Len equ $ - LABEL_SEG_CODE32

; END of [SECTION .s32]


; 16 位代码段. 由 32 位代码段跳入, 跳出后到实模式

[SECTION .s16code]

ALIGN    32

[BITS    16]

LABEL_SEG_CODE16:

    ; 跳回实模式:

    mov    ax, SelectorNormal

    mov    ds, ax

    mov    es, ax

    mov    fs, ax

    mov    gs, ax

    mov    ss, ax


    mov    eax, cr0

    and    al, 11111110b

    mov    cr0, eax


LABEL_GO_BACK_TO_REAL:

    jmp    0:LABEL_REAL_ENTRY    ; 段地址会在程序开始处被设置成正确的值


Code16Len    equ $ - LABEL_SEG_CODE16

```

```
; END of [SECTION .s16code]
```

3.2.2 主要代码结构

3.2.2.1 代码段[SECTION .s16]: 从实模式跳到保护模式

- 初始化 16 位代码段描述符
 - 初始化 32 位代码段描述符: 将[SECTION .S32]段的物理地址分三部分赋给描述符 DESC_CODE32
 - 初始化 数据段描述符
 - 初始化 堆栈段描述符
 - 为加载 GDT 做准备:将 GDT 的物理地址填充到 GdtPtr 中
 - 加载 GDT: lgdt [GdtPtr] #将 GdtPtr 加载到寄存器 gdtr 中
- 关中断

- 打开 A20 地址线: 通过操作端口 92h
- 准备切换到保护模式: 把寄存器 cr0 的第 0 位置为 1
- 进入保护模式: Jump

函数 LABEL_REAL_ENTRY: (从保护模式跳回到实模式前, 先回到这里)

- 程序重设各个段寄存器值, 使 ds、es、ss 指向 as
- 恢复 sp 的值
- 关闭 A20
- 打开中断
- 回到 DOS

3.2.2.2 代码段[SECTION .s32]: 32 位代码段, 从实模式跳入

- 初始化 ds、es、gs、ss: ds-指向数据段; es-指向新增的 5MB 的内存段; gs-指向显存; ss-指向测试段
- 显示一个字符串
- 调用读内存函数 TestRead
- 调用写函数 TestWrite

- 调用函数 TestRead
- 跳转到[SECTION .s16code]代码段
- TestRead 函数
- TestWrite 函数
- DispAL 函数:将 AL 中的字节用十六进制数表示出来，子的前景色是红色
- DispReturn 函数：模拟一个回车显示，让下一个字符显示在下一行的开头

3.2.2.3 代码段[SECTION .s16code]: 16 位代码段. 从 32 位代码段跳入, 跳出后到实模式

- 把选择子 SelectorNormal 加载到 ds、es、ss
- 清 cr0 的 PE 位，跳回实模式
- 跳转到代码段[SECTION .s16]中的函数 LABEL_REAL_ENTRY

3.2.3 调试过程

在工作目录中打开终端，输入 **bochs** 指令，等到 freedos 加载成功后，格式化 B 盘；输入 **format b** 指令，格式化完成后，将 pmtest2.asm 编译为 com 文件：

```
nasm pmtest2.asm -o pmtest2.com
```

紧接着，需要输入以下三条指令，将 pmtest2.com 复制到虚拟软盘 pm.img 中，所需要的指令如下：

```
sudo mount -o loop pm.img /mnt/floppy
```

```
sudo cp pmtest2.com /mnt/floppy
```

```
sudo umount /mnt/floppy
```

下一步需要重新回到 FreeDos 中（即原先打开的 Bochs 虚拟机），并且输入 **B:\pmtest2.com** 指令。至此，pmtest2.com 运行成功，如图 3.2 所示。

因为无法实现从 32 位代码段返回时 cs 高速缓存寄存器中的属性符合实模式的要求（实模式不能改变段属性）。

3.3 局部描述符表 LDT

3.3.1 关键代码

代码段 3.3 pmtest3.asm

```
[SECTION .gdt]

...

LABEL_DESC_LDT:    Descriptor    0,          LDTLen - 1, DA_LDT    ; LDT

...

[SECTION .s16]

...

    ; 初始化 LDT 在 GDT 中的描述符

    xor    eax, eax

    mov    ax, ds

    shl    eax, 4

    add    eax, LABEL_LDT

    mov    word [LABEL_DESC_LDT + 2], ax

    shr    eax, 16

    mov    byte [LABEL_DESC_LDT + 4], al

    mov    byte [LABEL_DESC_LDT + 7], ah

    ; 初始化 LDT 中的描述符

    xor    eax, eax

    mov    ax, ds

    shl    eax, 4

    add    eax, LABEL_CODE_A

    mov    word [LABEL_LDT_DESC_CODEA + 2], ax

    shr    eax, 16

    mov    byte [LABEL_LDT_DESC_CODEA + 4], al

    mov    byte [LABEL_LDT_DESC_CODEA + 7], ah

...
```



```

[SECTION .s32]; 32 位代码段. 由实模式跳入.

...

; Load LDT

mov ax, SelectorLDT

lldt ax

jmp SelectorLDTCodeA:0 ; 跳入局部任务

...

; LDT
[SECTION .ldt]
ALIGN 32
LABEL_LDT:
;                段基址      段界限      属性
LABEL_LDT_DESC_CODEA: Descriptor 0, CodeALen - 1, DA_C + DA_32 ; Code, 32 位

LDTLen equ $ - LABEL_LDT

; LDT 选择子
SelectorLDTCodeA equ LABEL_LDT_DESC_CODEA - LABEL_LDT + SA_TIL
; END of [SECTION .ldt]

; CodeA (LDT, 32 位代码段)
[SECTION .la]
ALIGN 32
[BITS 32]
LABEL_CODE_A:

mov ax, SelectorVideo

mov gs, ax ; 视频段选择子(目的)

```

```

mov edi, (80 * 12 + 0) * 2    ; 屏幕第 10 行, 第 0 列。

mov ah, 0Ch                   ; 0000: 黑底    1100: 红字

mov al, 'L'

mov [gs:edi], ax

; 准备经由 16 位代码段跳回实模式

jmp SelectorCode16:0

CodeALen    equ $ - LABEL_CODE_A

; END of [SECTION .la]

```

3.3.2 主要代码结构

3.3.2.1 代码段[SECTION .s16]: 从实模式跳到保护模式

- 初始化 16 位代码段描述符
- 初始化 32 位代码段描述符: 将[SECTION .s32]段的物理地址分三部分赋给描述符 DESC_CODE32
- 初始化 数据段描述符
- 初始化 堆栈段描述符
- 初始化 LDT 在 GDT 中的描述符
- 初始化 LDT 中的描述符
- 为加载 GDTR 做准备: 将 GDT 的物理地址填充到 GdtPtr 中
- 加载 GDTR: lgdt [GdtPtr] #将 GdtPtr 加载到寄存器 gdtr 中
- 关中断
- 打开 A20 地址线: 通过操作端口 92h
- 准备切换到保护模式: 把寄存器 cr0 的第 0 位置为 1
- 进入保护模式: Jump

函数 LABEL_REAL_ENTRY:

- 程序重设各个段寄存器值, 使 ds、es、ss 指向 as
- 恢复 sp 的值
- 关闭 A20
- 打开中断

- 回到 DOS

3.3.2.2 代码段[SECTION .s32]: 32 位代码段，从实模式跳入

- 初始化 ds、gs、ss
- 显示一个字符串
- 调用 DispReturn 函数
- 加载 LDT
- 跳入局部任务
- DispReturn 函数：模拟一个回车显示，让下一个字符显示在下一行的开头

3.3.2.3 代码段[SECTION .la]: LDT32 位代码段

- 将视频段选择子 SelectorVideo 的地址赋给 gs
- 设置输出位置为屏幕 10 行 0 列
- 设置输出为黑底红字
- 往显存中 edi 偏移处 ([gs:edi]地址处) 写入 L
- 跳到 16 位代码段[SECTION .s16code]

3.3.2.4 [SECTION .s16code]——16 位代码段。由 32 位代码段跳入，跳出后到实模式

- 把选择子 SelectorNormal 加载到 ds、es、ss、gs、fs
- 清 cr0 的 PE 位，跳回实模式
- 跳转到段[SECTION .s16]中的函数 LABEL_REAL_ENTRY

3.3.3 调试过程

调试过程同 3.2.3，只不过相应的代码变为：

nasm pmtest3.asm -o pmtest3.com

sudo cp pmtest3.com /mnt/floppy

至此，pmtest2.com 运行成功，如图 3.3 所示。



图 3.3 实验 3 结果

本实验 LDT 中的代码段非常简单，只是打印一个字符 L，因此，在 [SECTION .s32] 中打印完 “In Protect Mode Now.” 这个字符串之后，一个红色的字符 L 将会出现。

可以看到，在图 3.3 中，的确出现了 “In Protect Mode Now.” 字符串和一个红色的 L，这说明程序是正确的。

3.3.4 实验分析

在本实验中，笔者学习并了解了 LDT。简单地说，它是一种描述符表，与 GDT 差不多，只不过它的段选择子的 T1 位必须置为 1。在运用它时，必须先用 lldt 指令加载 ldtr，ldtr 的操作数是 GDT 中用来描述 LDT 的段描述符。

4 让操作系统走进保护模式

4.1 DOS 可以识别的引导盘

4.1.1 关键代码

代码段 4.1 boot.asm

```
;from chapter4\boot.asm

    jmp short LABEL_START      ; Start to boot.

    nop                        ; 这个 nop 不可少

; 下面是 FAT12 磁盘的头

BS_OEMName      DB 'ForrestY' ; OEM String, 必须 8 个字节
BPB_BytsPerSec  DW 512        ; 每扇区字节数
BPB_SecPerClus  DB 1          ; 每簇多少扇区
BPB_RsvdSecCnt  DW 1          ; Boot 记录占用多少扇区
BPB_NumFATs     DB 2          ; 共有多少 FAT 表
BPB_RootEntCnt  DW 224        ; 根目录文件数最大值

BPB_TotSec16    DW 2880       ; 逻辑扇区总数
BPB_Media       DB 0xF0       ; 媒体描述符
BPB_FATSz16     DW 9          ; 每 FAT 扇区数
BPB_SecPerTrk   DW 18         ; 每磁道扇区数
BPB_NumHeads    DW 2          ; 磁头数(面数)
BPB_HiddSec     DD 0          ; 隐藏扇区数
BPB_TotSec32    DD 0          ; wTotalSectorCount 为 0 时这个值记录扇区数
BS_DrvNum       DB 0          ; 中断 13 的驱动器号
BS_Reserved1    DB 0          ; 未使用
BS_BootSig      DB 29h        ; 扩展引导标记 (29h)
BS_VolID        DD 0          ; 卷序列号
BS_VolLab       DB 'OrangeS0.02'; 卷标, 必须 11 个字节
BS_FileSysType  DB 'FAT12'    ; 文件系统类型, 必须 8 个字节

;以上是 BPB (BIOS Parameter Block)
```

```

LABEL_START:

    mov ax, cs
    mov ds, ax
    mov es, ax
    Call DispStr      ; 调用显示字符串例程
    jmp $             ; 无限循环

DispStr:
    mov ax, BootMessage
    mov bp, ax        ; ES:BP = 串地址
    mov cx, 16        ; CX = 串长度
    mov ax, 01301h    ; AH = 13, AL = 01h
    mov bx, 000ch     ; 页号为 0(BH = 0) 黑底红字(BL = 0Ch,高亮)
    mov dl, 0
    int 10h           ; int 10h
    ret

BootMessage:    db "Hello, OS world!"
times 510-($-$$) db 0 ; 填充剩下的空间, 使生成的二进制代码恰好为 512 字节
dw 0xaa55       ; 结束标志

```

4.1.2 主要代码结构

- 使 ds 和 es 两个段寄存器指向与 cs 相同的段
- 调用 DispStr 函数显示字符串
- 无限循环

DispStr 函数:

- 设置 ES:BP = 串地址
- 设置 CX = 串长度
- 设置 AH = 13, AL = 01h
- 设置页号为 0(BH = 0) 黑底红字(BL = 0Ch,高亮)
- 10h 号中断

4.1.3 调试过程

调试过程基本同 1.1.4，输入命令：

```
nasm boot.asm -o boot.bin
```

```
dd if=boot.bin if=a.img bs=512 count=1 conv=notrunc
```

```
bochs
```

即可完成调试。结果如图 4.1 所示。

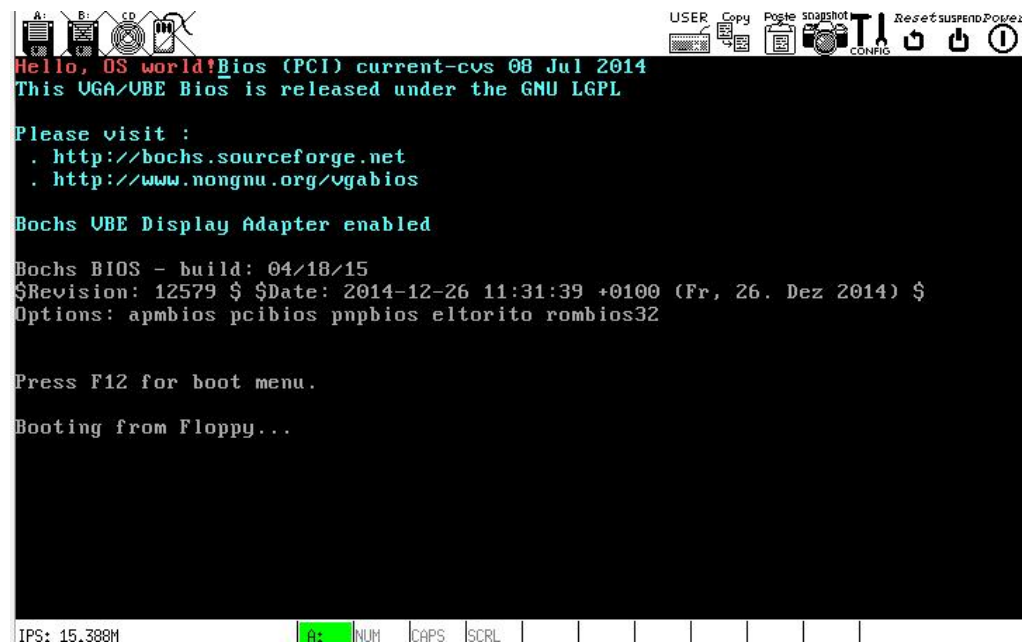


图 4.1 调试结果

可以看到，调试结果同实验 1.1，只不过这时笔者已经为引导扇区加入了 BPB 等头信息，使其可以被识别。

4.1.4 实验分析

本实验主要涉及的分析如下：

4.1.4.1 FAT12

FAT12 是 DOS 时代就开始使用的文件系统，直到现在仍然在软盘上使用。几乎所有的文件系统都会讲磁盘分为若干层次以方便组织和管理，这些层次包括：

- 扇区：磁盘上的最小数据单元。
- 簇：一个或多个扇区。
- 分区：通常指整个文件系统。

同时，FAT12 格式分为若干个扇区。引导扇区是整个磁盘的第 0 个扇区，在

这个扇区中有一个很重要的数据结构叫做 BPB，说明 FAT 的内容，之后则依次是 FAT1、FAT2、根目录区及数据区。

4.1.4.2 BPB

引导扇区需要有 BPB 等头信息才能被识别，在程序开头必须加上它。

4.1.4.3 突破 512 字节限制

进入保护模式是一件容易的事情，可是笔者要做的事情很多，总是局限于 512 字节的引导扇区之内肯定不够，必须有其他的办法。

作为解决办法，笔者可以利用软盘，其大小有 1.44M，对于操作系统雏形而言足够。因此可以再创建一个文件，将其通过引导扇区加载入内存，然后将控制权交给它，这样 512 字节的限制就解除了。

具体过程如下：引导->加载内核入内存->跳入保护模式->开始执行内核，这样的工作如果全部交给引导扇区做，空间坑不够，因此笔者将其交给 Loader 模块完成，引导扇区负责把 Loader 载入内存并将控制权移交给它，即可达成目的。

在这里，为了操作方便，不妨把软盘做成 FAT12 格式，这样对 Loader 和今后的内核的操作会非常简单易行。

4.2 实验 2 一个最简单的 Loader

4.2.1 关键代码

代码段 4.2 最简单的 Loader

```
;from chapter4\b\loader.asm
org 0100h

mov ax, 0B800h
mov gs, ax
mov ah, 0Fh          ; 0000: 黑底    1111: 白字
mov al, 'L'
mov [gs:((80 * 0 + 39) * 2)], ax ; 屏幕第 0 行, 第 39 列。

jmp $                ; 到此停住
```

代码段 4.3 寻找 Loader

```
; from chapter4\b\boot.asm
; 下面在 A 盘的根目录寻找 LOADER.BIN

mov word [wSectorNo], SectorNoOfRootDirectory
LABEL_SEARCH_IN_ROOT_DIR_BEGIN:

cmp word [wRootDirSizeForLoop], 0 ; \. 判断根目录区是不是已经读完
jz LABEL_NO_LOADERBIN ; / 如果读完表示没有找到 LOADER.BIN
dec word [wRootDirSizeForLoop] ; /
mov ax, BaseOfLoader
mov es, ax ; es <- BaseOfLoader
mov bx, OffsetOfLoader ; bx <- OffsetOfLoader
mov ax, [wSectorNo] ; ax <- Root Directory 中的某 Sector 号
mov cl, 1
call ReadSector

mov si, LoaderFileName ; ds:si -> "LOADER BIN"
mov di, OffsetOfLoader ; es:di -> BaseOfLoader:0100
```

```

cld

mov dx, 10h

LABEL_SEARCH_FOR_LOADERBIN:

cmp dx, 0                ;`. 循环次数控制,

jz  LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR; / 如果已经读完了一个 Sector,

dec dx                  ;/ 就跳到下一个 Sector

mov cx, 11

LABEL_CMP_FILENAME:

cmp cx, 0

jz  LABEL_FILENAME_FOUND ; 如果比较了 11 个字符都相等, 表示找到

dec cx

lodsb                   ; ds:si -> al

cmp al, byte [es:di]

jz  LABEL_GO_ON

jmp LABEL_DIFFERENT      ; 只要发现不一样的字符就表明本 DirectoryEntry

                           ; 不是我要找的 LOADER.BIN

LABEL_GO_ON:

inc di

jmp LABEL_CMP_FILENAME  ; 继续循环

LABEL_DIFFERENT:

and di, 0FFE0h          ; else`. di &= E0 为了让它指向本条目开头

add di, 20h              ;      |

mov si, LoaderFileName ;      | di += 20h  下一个目录条目

jmp LABEL_SEARCH_FOR_LOADERBIN; /

LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR:

add word [wSectorNo], 1

jmp LABEL_SEARCH_IN_ROOT_DIR_BEGIN

```

```

LABEL_NO_LOADERBIN:

    mov dh, 2                ; "No LOADER."

    call DispStr              ; 显示字符串

%ifdef _BOOT_DEBUG_

    mov ax, 4c00h            ; `

    int 21h                  ; / 没有找到 LOADER.BIN, 回到 DOS

%else

    jmp $                    ; 没有找到 LOADER.BIN, 死循环在这里

%endif

LABEL_FILENAME_FOUND:        ; 找到 LOADER.BIN 后便来到这里继续

    jmp $                    ; 代码暂时停在这里


times 510-($-$$) db 0        ; 填充剩下的空间, 使生成的二进制代码恰好为 512 字节

dw 0xaa55                   ; 结束标志

```

代码段 4.4 读软盘扇区

```

;from chapter4\b\boot.asm

;-----

; 函数名: ReadSector

;-----

; 作用:

;   从第 ax 个 Sector 开始, 将 cl 个 Sector 读入 es:bx 中

ReadSector:

    ; -----

    ; 怎样由扇区号求扇区在磁盘中的位置 (扇区号 -> 柱面号, 起始扇区, 磁头号)

    ; -----

    ; 设扇区号为 x

```

```

;                                r 柱面号 = y >> 1
;      x                        r 商 y |
; ----- => |                    L 磁头号 = y & 1
; 每磁道扇区数      |
;                                L 余 z => 起始扇区号 = z + 1

pushbp
mov bp, sp
sub esp, 2; 辟出两个字节的堆栈区域保存要读的扇区数: byte [bp-2]

mov byte [bp-2], cl

pushbx      ; 保存 bx
mov bl, [BPB_SecPerTrk]    ; bl: 除数
div bl      ; y 在 al 中, z 在 ah 中
inc ah      ; z ++
mov cl, ah   ; cl <- 起始扇区号
mov dh, al   ; dh <- y
shr al, 1    ; y >> 1 (y/BPB_NumHeads)
mov ch, al   ; ch <- 柱面号
and dh, 1    ; dh & 1 = 磁头号
pop bx       ; 恢复 bx
; 至此, "柱面号, 起始扇区, 磁头号" 全部得到

mov dl, [BS_DrvNum]      ; 驱动器号 (0 表示 A 盘)

.GoOnReading:
mov ah, 2      ; 读
mov al, byte [bp-2]    ; 读 al 个扇区
int 13h
jc .GoOnReading    ; 如果读取错误 CF 会被置为 1,
                  ; 这时就不停地读, 直到正确为止

add esp, 2
pop bp

```

`ret`

4.2.2 主要代码结构

4.2.2.1 文件 loader.asm

- 令 `gs` 指向 `0B8h00h` 处
- 设置黑底白字
- 设置 `AL='L'`
- 在 `gs` 偏移处(屏幕第 0 行 39 列)写入 `L`
- 无限循环

4.2.2.2 文件 boot.asm

- 使 `ds`、`es`、`ss` 三个段寄存器指向与 `cs` 相同的段
- 令栈指针 `sp` 指向栈底
- 软驱复位
- 在 A 盘的根目录寻找 `LOADER.BIN`: 遍历根目录取所有的扇区, 将每一个扇区加载到内存, 然后从中寻找文件名为 `Loader.bin` 的条目, 直到找到为止。
- `DispStr` 函数: 显示一个字符串
- `ReadSector` 函数: 从第 `ax` 个 Sector 开始, 将 `cl` 个 Sector 读入 `es:bx` 中
- 填充剩下的空间, 使生成的二进制代码恰好为 512 字节

4.2.3 调试过程

第一步, 编译文件生成二进制文件:

```
nasm boot.asm -o boot.bin
```

```
nasm loader.asm -o loader.bin
```

第二步, 先用 `bximage` 生成一个软盘映像, 然后输入代码将 `loader` 写入软盘:

```
dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc
```

```
sudo mount -o loop a.img /mnt/floppy
```

```
sudo cp loader.bin /mnt/floppy -v
```

```
sudo umount /mnt/floppy
```

第三步，这时不能直接启动（若直接启动则没有任何现象），因为仅仅是找到 Loader.bin 就停止在那里。这时应该使用 Bochs 的调试功能，在 Bochs 命令行内依次输入：

```

b 0x7c00    （在开始处设置断点）
c           （执行到断点）
n           （跳过 BPB）
u /45       （反汇编）
b 0x7cad    （在无限循环处设置断点）
c           （执行到断点）
x /13xcb es:di - 11 （容易发现 es: di 为我要找的文件名）

```

结果如图 4.2、图 4.3 和图 4.4 所示。

```

<bochs:1> b 0x7c00
<bochs:2> c
00000004662i[BIOS ] $Revision: 13752 $ $Date: 2019-12-30 14:16:18 +0100 (
Mon, 30 Dec 2019) $
00000318050i[KBD ] reset-disable command received
00000320814i[BIOS ] Starting rombios32
00000321255i[BIOS ] Shutdown flag 0
00000321887i[BIOS ] ram_size=0x02000000
00000322330i[BIOS ] ram_end=32MB
00000362973i[BIOS ] Found 1 cpu(s)
00000376661i[BIOS ] bios_table_addr: 0x000f9db8 end=0x000fcc00
00000601828i[XGUI ] charmap update. Font is 9 x 16
00000835579i[PCI ] i440FX PMC write to PAM register 59 (TLB Flush)

```

图 4.2 调试结果 1

```

<bochs:3> n
Next at t=14479644
(0) [0x000000007c3e] 0000:7c3e (unk. ctxt): mov ax, cs ; 8cc8
<bochs:4> u /45
0000000000007c3e: ( ): mov ax, cs ; 8cc8
0000000000007c40: ( ): mov ds, ax ; 8ed8
0000000000007c42: ( ): mov es, ax ; 8ec0
0000000000007c44: ( ): mov ss, ax ; 8ed0
0000000000007c46: ( ): mov sp, 0x7c00 ; bc007c
0000000000007c49: ( ): xor ah, ah ; 30e4

```

图 4.3 调试结果 2

```

<bochs:5> b 0x7cad
<bochs:6> c
(0) Breakpoint 2, 0x0000000000007cad in ?? ()
Next at t=14525424
(0) [0x000000007cad] 0000:7cad (unk. ctxt): jmp .-2 (0x00007cad) ; ebfe
<bochs:7> x /13xcb es:di - 11
[bochs]:
0x0000000000090120 <bogus+ 0>: L O A D E R
0x0000000000090128 <bogus+ 8>: B I N \0

```

图 4.4 调试结果 3

4.2.4 实验分析

在本实验中，笔者实现了将 Loader 加载进入内核，这用到了 BIOS 中断中的 int 13h，并且编写了读扇区和寻找 Loader.bin 的代码。

与此同时，笔者还尝试了 Bochs 的调试功能，用来验证程序的正确性，这同样是一个进步，而下一步就需要将 Loader.bin 加载如内存，并且移交控制权。

4.3 实验 3 向 Loader 交出控制权

4.3.1 关键代码

代码段 4.5 加载 Loader

```
;from chapter4\c\boot.asm

LABEL_FILENAME_FOUND:                ; 找到 LOADER.BIN 后便来到这里继续

    mov ax, RootDirSectors

    and di, 0FFE0h                    ; di -> 当前条目的开始

    add di, 01Ah                      ; di -> 首 Sector

    mov cx, word [es:di]

    pushcx                            ; 保存此 Sector 在 FAT 中的序号

    add cx, ax

    add cx, DeltaSectorNo             ; cl <- LOADER.BIN 的起始扇区号(0-based)

    mov ax, BaseOfLoader

    mov es, ax                        ; es <- BaseOfLoader

    mov bx, OffsetOfLoader            ; bx <- OffsetOfLoader

    mov ax, cx                        ; ax <- Sector 号

LABEL_GOON_LOADING_FILE:

    pushax                            ; \

    pushbx                            ; |

    mov ah, 0Eh                       ; | 每读一个扇区就在 "Booting " 后面

    mov al, '.'                       ; | 打一个点, 形成这样的效果:

    mov bl, 0Fh                       ; | Booting .....

    int 10h                           ; |

    pop bx                            ; |

    pop ax                            ; /

    mov cl, 1

    call ReadSector
```



```

    pop ax          ; 取出此 Sector 在 FAT 中的序号

    call GetFATEntry

    cmp ax, 0FFFh

    jz LABEL_FILE_LOADED

    pushax          ; 保存 Sector 在 FAT 中的序号

    mov dx, RootDirSectors

    add ax, dx

    add ax, DeltaSectorNo

    add bx, [BPB_BytsPerSec]

    jmp LABEL_GOON_LOADING_FILE
LABEL_FILE_LOADED:

    mov dh, 1        ; "Ready."

    call DispStr      ; 显示字符串

;

*****

*****

    jmp BaseOfLoader:OffsetOfLoader; 这一句正式跳转到已加载到内存
                                     ; 存中的 LOADER.BIN 的开始处，
                                     ; 开始执行 LOADER.BIN 的代码。
                                     ; Boot Sector 的使命到此结束。

```

代码段 4.6 由扇区号求 FAT 项的值

```

;from chapter4\c\boot.asm

;-----

; 函数名: GetFATEntry

;-----

; 作用:

;   找到序号为 ax 的 Sector 在 FAT 中的条目, 结果放在 ax 中

;   需要注意的是, 中间需要读 FAT 的扇区到 es:bx 处, 所以函数一开始保存了 es 和 bx

```

GetFATEntry:

```
pushes
pushbx
pushax
mov ax, BaseOfLoader; `
sub ax, 0100h; | 在 BaseOfLoader 后面留出 4K 空间用于存放 FAT
mov es, ax      ;/
pop ax
mov byte [bOdd], 0
mov bx, 3
mul bx          ; dx:ax = ax * 3
mov bx, 2
div bx          ; dx:ax / 2 ==> ax <- 商, dx <- 余数
cmp dx, 0
jz LABEL_EVEN
mov byte [bOdd], 1
```

LABEL_EVEN;;偶数

; 现在 ax 中是 FATEntry 在 FAT 中的偏移量,下面来

; 计算 FATEntry 在哪个扇区中(FAT 占用不止一个扇区)

```
xor dx, dx
```

```
mov bx, [BPB_BytsPerSec]
```

```
div bx ; dx:ax / BPB_BytsPerSec
```

; ax <- 商 (FATEntry 所在的扇区相对于 FAT 的扇区号)

; dx <- 余数 (FATEntry 在扇区内的偏移)。

```
pushdx
```

```
mov bx, 0; bx <- 0 于是, es:bx = (BaseOfLoader - 100):00
```

```
add ax, SectorNoOfFAT1; 此句之后的 ax 就是 FATEntry 所在的扇区号
```

```
mov cl, 2
```

```
call ReadSector; 读取 FATEntry 所在的扇区, 一次读两个, 避免在边界
```

; 发生错误, 因为一个 FATEntry 可能跨越两个扇区

```

    pop dx

    add bx, dx

    mov ax, [es:bx]

    cmp byte [bOdd], 1

    jnz LABEL_EVEN_2

    shr ax, 4

LABEL_EVEN_2:

    and ax, 0FFFh

LABEL_GET_FAT_ENRY_OK:


    pop bx

    pop es

    ret

```

4.3.2 主要代码结构

- 使 ds、es、ss 三个段寄存器指向与 cs 相同的段
- 令栈指针 sp 指向栈底
- 清屏并显示字符串 Booting
- 软驱复位
- 在 A 盘的根目录寻找 LOADER.BIN: 遍历根目录取所有的扇区, 将每一个扇区加载到内存, 然后从中寻找文件名为 Loader.bin 的条目, 直到找到为止。
- 每读一个扇区就“Booting”后面打一个点, 形成这样的效果:Booting
- 显示字符串“Ready”
- 跳转到已加载到内存中的 LOADER.BIN 的开始处, 开始执行 LOADER.BIN 的代码, Boot Sector 的使命到此结束。
- Dispstr 函数: 显示一个字符串
- ReadSector 函数: 从第 ax 个 Sector 开始, 将 cl 个 Sector 读入 es:bx 中
- GetFATEntry 函数: 找到序号为 ax 的 Sector 在 FAT 中的条目, 结果放在 ax 中, 需要注意的是, 中间需要读 FAT 的扇区到 es:bx 处, 所以函数一开

始保存了 es 和 bx

4.3.3 调试过程

首先，编译文件生成二进制文件：

```
nasm boot.asm -o boot.bin
```

```
nasm loader.asm -o loader.bin
```

然后，先用 bimage 生成一个软盘印象，然后输入代码将 loader 写入软盘：

```
dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc
```

```
sudo mount -o loop a.img /mnt/floppy
```

```
sudo cp loader.bin /mnt/floppy -v
```

```
sudo umount /mnt/floppy
```

最后，进入 Bochs，输入 c 并按下回车键，开始调试。结果如图 4.5 所示。

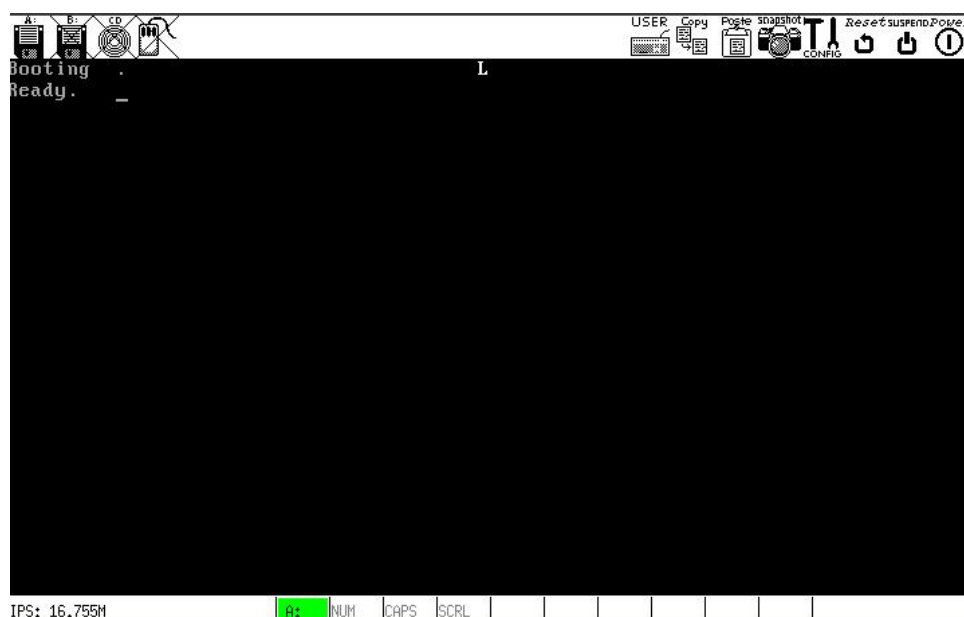


图 4.5 调试结果

可以看到，屏幕上出现了 Booting 和 Ready 的字样，代表程序进入引导扇区和进入引导扇区结束，紧接着，程序跳转入 Loader，在屏幕最上方的正中间打印了一个白色的“L”字，调试结果说明实验成功。

4.3.4 实验分析

在本实验中，笔者实现了 Boot 的代码，使其可以从软盘中读出 Loader.bim 文件，并且加载入内核，同时移交控制权。

现在，只要一个.COM 文件中不含有 DOS 系统调用，笔者就可以将它当成 Loader 使用，也就是说，现在的程序已经可以被看作是一个在保护模式下执行的“操作系统”了。

但是，目前的 Loader 仅仅只是一个 Loader，它不是操作系统内核，也不能当做操作系统内核，因为笔者希望操作系统内核至少可以在 Linux 下用 GCC 编译链接，摆脱汇编语言，而 Loader 则至少要做两件事：将内核 Kernel 加载入内核、跳入保护模式，这也是下一章的内容之一。

5 内核雏形

5.1 在 Linux 下用汇编写 Hello World

5.1.1 关键代码

代码段 5.1 hello.asm

```
; from chapter5\a\hello.asm
; 编译链接方法
; (ld 的'-s'选项意为“strip all”)
;
; $ nasm -f elf hello.asm -o hello.o
; $ ld -s hello.o -o hello
; $ ./hello
; Hello, world!
; $

[section .data] ; 数据在此

strHello db "Hello, world!", 0Ah
STRLEN equ $ - strHello

[section .text] ; 代码在此

global _start ; 我必须导出 _start 这个入口，以便让链接器识别

_start:

    mov edx, STRLEN

    mov ecx, strHello

    mov ebx, 1

    mov eax, 4 ; sys_write

    int 0x80 ; 系统调用
```

```
mov ebx, 0

mov eax, 1      ; sys_exit

int  0x80      ; 系统调用
```

5.1.2 主要代码结构

- 让 `_start` 符号成为可见的标识符，这样链接器就知道跳转到程序中的什么地方并开始执行程序

`_start` 函数:

- 设置字符串长度
- 设置要显示的字符串
- 设置文件描述符(stdout)
- 系统调用输出字符串(sys_write)
- 退出函数

5.1.3 调试过程

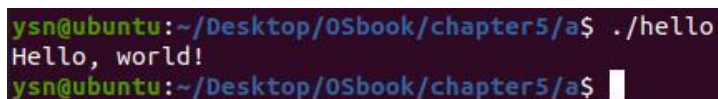
依次输入以下指令，完成调试:

```
nasm -f elf hello.asm -o hello.o
```

```
ld -s hello.o -o hello
```

```
./hello
```

调试结果如图 5.1 所示:



```
ysn@ubuntu:~/Desktop/OSbook/chapter5/a$ ./hello
Hello, world!
ysn@ubuntu:~/Desktop/OSbook/chapter5/a$
```

图 5.1 用汇编语言写 Hello World 调试结果

5.1.4 实验分析

在本实验中，程序定义了两个节（Section），一个放数据，一个放代码。在代码中值得注意的一点是，入口点默认的是“`_start`”，笔者不但要定义它，而且要通过 `global` 这个关键字将它导出，这样连接程序才能找到它，至于代码本身，则利用了两个系统调用。

5.2 再进一步，汇编和 C 同步使用

5.2.1 关键代码

代码段 5.2 foo.asm

```
; from chapter5\b\foo.asm
; 编译链接方法
; (ld 的'-s'选项意为“strip all”)
;
; $ nasm -f elf foo.asm -o foo.o
; $ gcc -c bar.c -o bar.o
; $ ld -s hello.o bar.o -o foobar
; $ ./foobar
; the 2nd one
; $

extern choose ; int choose(int a, int b);

[section .data] ; 数据在此

num1st      dd 3
num2nd      dd 4

[section .text] ; 代码在此

global _start ; 我必须导出 _start 这个入口，以便让链接器识别。
global myprint ; 导出这个函数为了让 bar.c 使用

_start:

    pushdword [num2nd] ; `
    pushdword [num1st] ; |
```



```

call choose      ; | choose(num1st, num2nd);

add esp, 8       ; /

mov ebx, 0

mov eax, 1        ; sys_exit

int 0x80         ; 系统调用

```

```

; void myprint(char* msg, int len)

```

```

myprint:

```

```

    mov edx, [esp + 8] ; len

    mov ecx, [esp + 4] ; msg

    mov ebx, 1

    mov eax, 4          ; sys_write

    int 0x80           ; 系统调用

    ret

```

代码段 5.3 bar.c

```

; from chapter5\b\bar.c

void myprint(char* msg, int len);

int choose(int a, int b)
{
    if(a >= b){
        myprint("the 1st one\n", 13);
    }
    else{
        myprint("the 2nd one\n", 13);
    }

    return 0;
}

```

5.2.2 主要代码结构

5.2.2.1 [section .text]代码段

- 让 `_start` 和 `myprint` 符号成为可见的标识符，这样链接器就知道跳转到程序中的什么地方并开始执行程序。

`_start` 函数:

- 把 `num2nd`、`num1st` 先后压入栈
- 调用外部函数 `choose`
- 栈指针加 8
- 设置参数文件描述符(`stdout`)
- 设置系统调用号(`sys_write`)
- 进行系统调用

`myprint` 函数:

- 设置参数 `len`
- 设置参数 `msg`
- 设置参数文件描述符(`stdout`)
- 设置系统调用号(`sys_write`)
- 调用内核功能
- 返回

5.2.2.2 文件 `bar.c`

- 外部函数 `myprint()`的声明

`choose (a,b)` 函数:

- 如果 `a>=b`:调用 `myprint` 输出"the 1st one"
否则: 调用 `myprint` 输出"the 2nd one"

5.2.3 调试过程

输入以下指令以进行编译链接和执行:

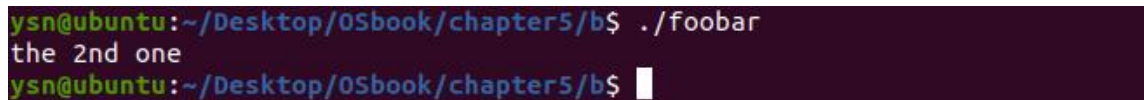
```
nasm -f elf -o foo.o foo.asm
```

```
gcc -m32 -c -o bar.o bar.c
```

```
ld -m elf_i386 -s -o foobar foo.o bar.o
```

./foobar

调试结果如图 5.2 所示：



```
ysn@ubuntu:~/Desktop/OSbook/chapter5/b$ ./foobar
the 2nd one
ysn@ubuntu:~/Desktop/OSbook/chapter5/b$
```

图 5.2 foobar 执行结果

在本实验中，定义了 num1=3，num2=4，程序输出大的那个数的结果，可以看到，程序输出了“the 2nd one”的字样，成功地完成了任务。

5.2.4 实验分析

在本实验中，笔者实现了汇编语言与 c 语言的共同使用，将它们编译为 elf 格式的文件并最终执行。

其中，源代码包含两个文件：foo.asm 和 bar.c。程序入口_start 在 foo.asm 中，一开始程序会调用 bar.c 中的函数 choose，choose()会比较传入的两个参数，根据比较结果的不同打印出不同的字符串。打印字符串的工作是由 foo.asm 中的函数 myprint()来完成的。

其中，在 c 语言中调用汇编语言的函数，需要在汇编语言中把该函数定义为 global，而如果汇编语言要用到 C 语言中的函数，需要将相应的函数定义为 extern，同时参数遵循 C 调用约定，后面的参数先入栈，并且由调用者清理堆栈。

下面再谈 ELF（Executable and Linkable Format）文件格式，ELF 文件由四个部分组成：ELF 头（ELF header）、程序头表（Program header table）、节（Sections）和节头标（Section header table），定义了有关 ELF 格式的有关信息，并且可以使汇编语言和 C 语言串联起来。

5.3 从 Loader 到内核

5.3.1 关键代码

代码段 5.4 共享常量所在的文件 fat12hdr.inc

```
;from chapter5\c\fat12hdr.inc
; FAT12 磁盘的头
; -----
BS_OEMName      DB 'ForrestY' ; OEM String, 必须 8 个字节

BPB_BytsPerSec  DW 512        ; 每扇区字节数
BPB_SecPerClus  DB 1          ; 每簇多少扇区
BPB_RsvdSecCnt  DW 1          ; Boot 记录占用多少扇区
BPB_NumFATs     DB 2          ; 共有多少 FAT 表
BPB_RootEntCnt  DW 224        ; 根目录文件数最大值
BPB_TotSec16    DW 2880       ; 逻辑扇区总数
BPB_Media       DB 0xF0       ; 媒体描述符
BPB_FATSz16     DW 9          ; 每 FAT 扇区数
BPB_SecPerTrk   DW 18         ; 每磁道扇区数
BPB_NumHeads    DW 2          ; 磁头数(面数)
BPB_HiddSec     DD 0           ; 隐藏扇区数
BPB_TotSec32    DD 0           ; 如果 wTotalSectorCount 是 0 由这个值记录扇区数

BS_DrvNum       DB 0           ; 中断 13 的驱动器号
BS_Reserved1    DB 0           ; 未使用
BS_BootSig      DB 29h        ; 扩展引导标记 (29h)
BS_VolIDDD      DB 0           ; 卷序列号
BS_VolLab       DB 'OrangeS0.02'; 卷标, 必须 11 个字节
BS_FileSysType  DB 'FAT12 '    ; 文件系统类型, 必须 8 个字节
; -----
```

```

; -----
; 基于 FAT12 头的一些常量定义，如果头信息改变，下面的常量可能也要做相应改变
; -----
; BPB_FATs16
FATsSz      equ 9

; 根目录占用空间:
; RootDirSectors = ((BPB_RootEntCnt*32)+(BPB_BytsPerSec-1))/BPB_BytsPerSec
; 但如果按照此公式代码过长，故定义此宏
RootDirSectors equ 14

; Root Directory 的第一个扇区号 = BPB_RsvdSecCnt + (BPB_NumFATs * FATsSz)
SectorNoOfRootDirectory equ 19

; FAT1 的第一个扇区号 = BPB_RsvdSecCnt
SectorNoOfFAT1      equ 1

; DeltaSectorNo = BPB_RsvdSecCnt + (BPB_NumFATs * FATsSz) - 2
; 文件的开始 Sector 号 = DirEntry 中的开始 Sector 号 + 根目录占用 Sector 数目
;
; + DeltaSectorNo
DeltaSectorNo      equ 17

```

代码段 5.5 loader.asm

```

; from chapter5\c\loader.asm
org 0100h

BaseOfStack      equ 0100h

BaseOfKernelFile equ 08000h ; KERNEL.BIN 被加载到的位置 ---- 段地址
OffsetOfKernelFile equ 0h ; KERNEL.BIN 被加载到的位置 ---- 偏移地址

```

```

    jmp LABEL_START          ; Start

; 下面是 FAT12 磁盘的头, 之所以包含它是因为下面用到了磁盘的一些信息
#include "fat12hdr.inc"

LABEL_START:                ; <--- 从这里开始 *****

    mov ax, cs
    mov ds, ax
    mov es, ax
    mov ss, ax
    mov sp, BaseOfStack

    mov dh, 0                ; "Loading  "
    call DispStr              ; 显示字符串

; 下面在 A 盘的根目录寻找 KERNEL.BIN
    mov word [wSectorNo], SectorNoOfRootDirectory

    xor ah, ah                ; `
    xor dl, dl                ; | 软驱复位
    int 13h ;/

LABEL_SEARCH_IN_ROOT_DIR_BEGIN:

    cmp word [wRootDirSizeForLoop], 0 ; `
    jz LABEL_NO_KERNELBIN    ; | 判断根目录区是不是已经读完,
    dec word [wRootDirSizeForLoop] ;/ 读完表示没有找到 KERNEL.BIN
    mov ax, BaseOfKernelFile
    mov es, ax                ; es <- BaseOfKernelFile
    mov bx, OffsetOfKernelFile ; bx <- OffsetOfKernelFile

```

```
mov ax, [wSectorNo] ; ax <- Root Directory 中的某 Sector 号
```

```
mov cl, 1
```

```
call ReadSector
```

```
mov si, KernelFileName ; ds:si -> "KERNEL BIN"
```

```
mov di, OffsetOfKernelFile
```

```
cld
```

```
mov dx, 10h
```

LABEL_SEARCH_FOR_KERNELBIN:

```
cmp dx, 0 ; \.
```

```
jz LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR; | 循环次数控制, 如果已经读完
```

```
dec dx ;/ 了一个 Sector, 就跳到下一个
```

```
mov cx, 11
```

LABEL_CMP_FILENAME:

```
cmp cx, 0 ; \.
```

```
jz LABEL_FILENAME_FOUND ; | 循环次数控制, 如果比较了 11 个字符都
```

```
dec cx ;/ 相等, 表示找到
```

```
lodsb ; ds:si -> al
```

```
cmp al, byte [es:di] ; if al == es:di
```

```
jz LABEL_GO_ON
```

```
jmp LABEL_DIFFERENT
```

LABEL_GO_ON:

```
inc di
```

```
jmp LABEL_CMP_FILENAME ; 继续循环
```

LABEL_DIFFERENT:

```
and di, 0FFE0h ; else \. 让 di 是 20h 的倍数
```

```
add di, 20h ; |
```

```
mov si, KernelFileName ; | di += 20h 下一个目录条目
```

```
jmp LABEL_SEARCH_FOR_KERNELBIN; /
```

```

LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR:

    add word [wSectorNo], 1

    jmp LABEL_SEARCH_IN_ROOT_DIR_BEGIN

LABEL_NO_KERNELBIN:

    mov dh, 2                ; "No KERNEL."
    call DispStr             ; 显示字符串

#ifdef _LOADER_DEBUG_

    mov ax, 4c00h            ; `.`
    int 21h                  ; / 没有找到 KERNEL.BIN, 回到 DOS
#else

    jmp $                    ; 没有找到 KERNEL.BIN, 死循环在这里
#endif

LABEL_FILENAME_FOUND:        ; 找到 KERNEL.BIN 后便来到这里继续

    mov ax, RootDirSectors

    and di, 0FFF0h           ; di -> 当前条目的开始

    pusheax

    mov eax, [es : di + 01Ch] ; `.`
    mov dword [dwKernelSize], eax ; / 保存 KERNEL.BIN 文件大小
    pop eax

    add di, 01Ah             ; di -> 首 Sector

    mov cx, word [es:di]

    pushcx                   ; 保存此 Sector 在 FAT 中的序号
    add cx, ax

    add cx, DeltaSectorNo    ; cl <- LOADER.BIN 的起始扇区号(0-based)

    mov ax, BaseOfKernelFile

```



```

mov es, ax          ; es <- BaseOfKernelFile

mov bx, OffsetOfKernelFile ; bx <- OffsetOfKernelFile

mov ax, cx          ; ax <- Sector 号

```

LABEL_GOON_LOADING_FILE:

```

pushax             ; `
pushbx             ; |
mov ah, 0Eh         ; | 每读一个扇区就在 "Loading " 后面
mov al, '.'         ; | 打一个点, 形成这样的效果:
mov bl, 0Fh         ; | Loading .....
int 10h            ; |
pop bx             ; |
pop ax             ; /

```

```

mov cl, 1
call ReadSector

pop ax             ; 取出此 Sector 在 FAT 中的序号
call GetFATEntry

cmp ax, 0FFFh
jz LABEL_FILE_LOADED

pushax             ; 保存 Sector 在 FAT 中的序号
mov dx, RootDirSectors
add ax, dx
add ax, DeltaSectorNo
add bx, [BPB_BytsPerSec]
jmp LABEL_GOON_LOADING_FILE

```

LABEL_FILE_LOADED:

```

call KillMotor     ; 关闭软驱马达

```

```

mov dh, 1          ; "Ready."
call DispStr       ; 显示字符串

jmp $

```

代码段 5.6 内核雏形 kernel.asm

```

;from chapter5\c\kernel.asm
; 编译链接方法
; $ nasm -f elf kernel.asm -o kernel.o
; $ ld -s kernel.o -o kernel.bin    #-s'选项意为“strip all”

[section .text] ; 代码在此

global _start   ; 导出 _start

_start:        ; 跳到这里来的时候, 我假设 gs 指向显存

    mov ah, 0Fh          ; 0000: 黑底    1111: 白字
    mov al, 'K'
    mov [gs:((80 * 1 + 39) * 2)], ax    ; 屏幕第 1 行, 第 39 列。
    jmp $

```

5.3.2 主要代码结构

5.3.2.1 文件 loader.asm

- 使 ds、es、ss 三个段寄存器指向与 cs 相同的段
- 令栈指针 sp 指向栈底
- 清屏并显示字符串 Loading
- 在 A 盘的根目录寻找 KERNEL.BIN: 遍历根目录取所有的扇区, 将每一个扇区加载到内存, 然后从中寻找文件名为 kernel.bin 的条目, 直到找到为止。
- 每读一个扇区就在 "Loading " 后面打一个点, 形成这样的效果:Loading
- 关闭软驱马达
- 显示字符串“Ready”

- 无限循环

5.3.2.2 文件 Kernel.asm

- 导出 `_start`
- 跳到 `_start`, 假设 `gs` 指向显存
- 设置黑底白字
- 设置 `AL='K'`
- 在 `gs` 偏移处(屏幕第 1 行 39 列)写入 `L`
- 无限循环

5.3.3 调试过程

第一步，输入以下代码进行编译：

```
nasm boot.asm -o boot.bin
```

```
nasm loader.asm -o loader.bin
```

```
nasm -f elf -o kernel.o kernel.asm
```

```
ld -m elf_i386 -s -o kernel.bin kernel.o
```

第二步，输入以下代码进行调试：

```
dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc
```

```
sudo mount -o loop a.img /mnt/floppy/
```

```
sudo cp loader.bin /mnt/floppy/ -v
```

```
sudo cp kernel.bin /mnt/floppy/ -v
```

```
sudo umount /mnt/floppy/
```

第三步，进入 Bochs，输入 `c` 并回车开始调试。结果如图 5.3 所示。



图 5.3 调试结果

可以看到，在上一个实验的基础上，这次的实验结果多出了“Loading.....”及“Ready.”这样的两行，说明已经载入了内核，并且由 Loader 读取了一个扇区。不过，由于目前除了把内核加载到内存之外没有做其他任何工作，所以除了能看

到“Ready.”的字样之外，并没有其他现象出现。

5.3.4 实验分析

在本实验中，笔者成功地加载了内核 `Kernel`，虽然这是 `Kernel` 中还没有其他内容，但已经是一个内核的雏形，之后对于内核，还可以脱离汇编语言，使用 C 语言进行编程，是一个非常大的进步。

6 进程

6.1 最简单的进程

6.1.1 关键代码

代码段 6.1 时钟中断程序

```
; from chapter6\kernel\kernel.asm  
  
ALIGN 16  
  
hwint00: ; Interrupt routine for irq 0 (the clock).  
  
iretd
```

代码段 6.2 restart 函数

```
; from chapter6\kernel\kernel.asm  
  
restart:  
  
    mov esp, [p_proc_ready]  
  
    lldt [esp + P_LDT_SEL]  
  
    lea eax, [esp + P_STACKTOP]  
    mov dword [tss + TSS3_S_SP0], eax  
  
  
    pop gs  
  
    pop fs  
  
    pop es  
  
    pop ds  
  
    popad  
  
  
    add esp, 4  
  
  
    iretd
```

代码段 6.3 kernel_main 函数

```
; from chapter6\kernel\main.c  
  
PUBLIC int kernel_main()  
  
{
```

```

disp_str("----\"kernel_main\" begins----\n");

PROCESS* p_proc = proc_table;

p_proc->ldt_sel = SELECTOR_LDT_FIRST;

memcpy(&p_proc->ldts[0], &gdt[SELECTOR_KERNEL_CS>>3], sizeof(DESCRIPTOR));
p_proc->ldts[0].attr1 = DA_C | PRIVILEGE_TASK << 5; // change the DPL
memcpy(&p_proc->ldts[1], &gdt[SELECTOR_KERNEL_DS>>3], sizeof(DESCRIPTOR));
p_proc->ldts[1].attr1 = DA_DRW | PRIVILEGE_TASK << 5; // change the DPL


p_proc->regs.cs = (0 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
p_proc->regs.ds = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
p_proc->regs.es = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
p_proc->regs.fs = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
p_proc->regs.ss = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
p_proc->regs.gs = (SELECTOR_KERNEL_GS & SA_RPL_MASK) | RPL_TASK;
p_proc->regs.eip= (u32)TestA;
p_proc->regs.esp= (u32) task_stack + STACK_SIZE_TOTAL;
p_proc->regs.eflags = 0x1202; // IF=1, IOPL=1, bit 2 is always 1.


p_proc_ready = proc_table;

restart();

while(1){
}

```

代码段 6.4 TestA 函数

```

; from chapter6\kernel\main.c

void TestA()
{
    int i = 0;

```

```

while(1){
    disp_str("A");
    disp_int(i++);
    disp_str(".");
    delay(1);
}
}

```

代码段 6.5 delay 函数

```

; from chapter6\lib\klib.c
PUBLIC void delay(int time)
{
    int i, j, k;
    for (k = 0; k < time; k++) {
        for (i = 0; i < 10; i++) {
            for (j = 0; j < 10000; j++) {}
        }
    }
}
}

```

6.1.2 代码主要结构

Kernel 启动流程:

- 准备好进程体 TestA
- 对进程表、TSS 进行初始化
- 准备进程表
- 时钟中断处理实现 ring0 到 ring1 的跳转
- 开始进程
- 结束任务

6.1.3 调试过程

由于本章之前引入了自动化编译和链接的工具 GNU Make, 因此只需要进入

工作目录，输入 **make image** 命令进行自动编译连接，紧接着便可输入 **bochs** 命令开始调试。

但是，教材配套代码是存在问题的，如果直接 **make image**，调试结果会出现如图 6.1 所示的错误。

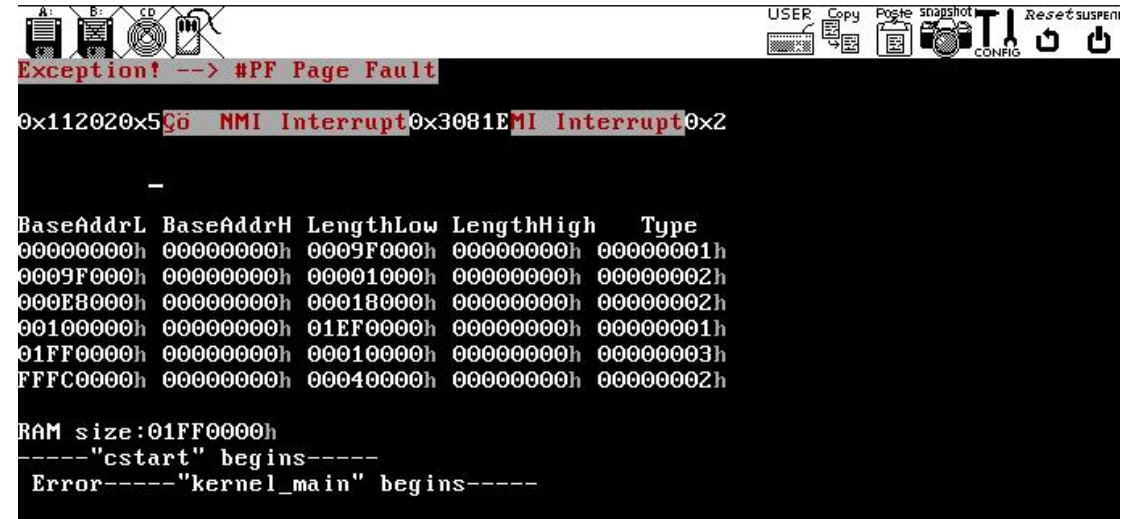


图 6.1 错误调试结果

根据笔者的分析，问题出在代码 `lib/klib.asm` 的 `disp_str` 函数存在部分代码缺失，故函数 `disp_str` 应作如下修改（标红语句为新增代码）：

```
; from chapter6\lib\klib.asm
disp_str:
    push ebp
    mov ebp, esp
    push ebx
    push esi
    push edi
    mov esi, [ebp + 8] ; pszInfo
    mov edi, [disp_pos]
    mov ah, 0Fh
.1:
    lodsb
    test al, al
    jz .2
```



```

    cmp al, 0Ah ; 是回车吗?
    jnz .3
    pusheax
    mov eax, edi
    mov bl, 160
    div bl
    and eax, 0FFh
    inc eax
    mov bl, 160
    mul bl
    mov edi, eax
    pop eax
    jmp .1
.3:
    mov [gs:edi], ax
    add edi, 2
    jmp .1
.2:
    mov [disp_pos], edi
    pop edi
    pop esi
    pop ebx
    pop ebp
    ret

```

完成代码修改后，重新输入 **make image** 命令，再输入 **bochs** 命令开始调试，得到结果如图 6.2 所示。（后续碰到相同问题，均采用此方法解决，不再赘述）

```

Booting .....
Ready.

Loading .....
Ready.

BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0009F000h 00000000h 00000001h
0009F000h 00000000h 00001000h 00000000h 00000002h
000E8000h 00000000h 00018000h 00000000h 00000002h
00100000h 00000000h 01EF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFC0000h 00000000h 00040000h 00000000h 00000002h

RAM size:01FF0000h
---->"cstart" begins-----
---->"cstart" finished-----
---->"kernel_main" begins-----
A0x0.A0x1.A0x2.A0x3.A0x4.A0x5.A0x6.A0x7.A0x8.A0x9.A0xA.A0xB.A0xC.A0xD.A0xE.A0xF.
A0x10.A0x11.A0x12.A0x13.A0x14.A0x15.A0x16.A0x17.A0x18.A0x19.A0x1A.A0x1B.A0x1C.A0
x1D.A0x1E.A0x1F.A0x20.A0x21.A0x22.A0x23.A0x24.A0x25.A0x26.A0x27.A0x28.A0x29.A0x2
A.A0x2B.A0x2C.A0x2D.A0x2E.A0x2F.A0x30.A0x31.A0x32.A0x33.A0x34.A0x35.A0x36.A0x37.
A0x38.A0x39.A0x3A.A0x3B.A0x3C.A0x3D.A0x3E.A0x3F.A0x40.A0x41.A0x42.A0x43.A0x44.A0
x45.A0x46.A0x47.A0x48.A0x49.A0x4A.A0x4B.A0x4C.A0x4D.A0x4E.A0x4F.A0x50.A0x51.A0x5
2.A0x53.A0x54.A0x55.A0x56.A0x57.A0x58.A0x59.A0x5A.A0x5B.A0x5C.A0x5D.A0x5E.A0x5F.
IPS: 8,841M
A: NUM CAPS SCRL

```

图 6.2 进程调试结果

由调试结果，笔者看到了不断出现的字符“A”和不断增加的数字，这代表进程已经开始运行，调试成功了。

虽然这是一个普通不过的函数在运行，但却有着不同寻常的意义，这意味着笔者实现了 ring0 到 ring1 的跳转，更进一步，这意味着进程在运行，而这一切意味着笔者编写的代码已经可以称之为一个操作系统了，因为它已经有了“进程”，尽管还非常简陋。

6.1.4 实验分析

在本实验中，笔者为操作系统添加了进程，而为了实现进程功能，还实现了以下的部分：

时钟中断处理实现，用以实现 ring0 到 ring1 的跳转；

全新的进程表、进程体、TSS 结构和更新过的 GDT 结构；

准备一个小的进程体；

对进程表、TSS 进行初始化.....

在一切准备完毕后，便可通过 restart 跳入进程，最终可以看到运行起来的进程。

而回顾整个程序，第一个进程的启动过程主要分为以下四步：

第一步，准备好进程体（本例中为 TestA()）；

第二步，初始化 GDT 中的 TSS 和 LDT 两个描述符，并且初始化 TSS（在 `init_port()` 中完成）；

第三步，准备进程表（在 `kernel_main()` 中完成）；

第四步，完成跳转，实现 ring0->ring1. 开始进程（`kernel.asm` 中的 `restart`）。

不过，笔者的进程和进程表仍然非常简单，第一是进程表不能保存并回复任务状态，第二是进程体开启后便不能停止，因为没有开启时钟中断，不能胜任多进程的任务，因此笔者会在下面的实验 2 及实验 3 对其改进与完善。

让操作系统走进保护模式，另外建立了一个文件，将其通过引导扇区加载如内存，引导扇区负责把 `Loader` 加载进内存，其他工作：跳入保护模式、开始执行内核等由 `Loader` 模块去做，这样能突破 512 字节的限制，灵活很多。

6.2 丰富中断处理程序

6.2.1 关键代码

代码段 6.6 打开时钟中断

```
; from chapter6\b\kernel\i8259.c  
  
out_byte(INT_M_CTLMASK, 0xFE); // Master 8259, OCW1.  
  
out_byte(INT_S_CTLMASK, 0xFF); // Slave 8259, OCW1.
```

代码段 6.7 设置 EOI

```
; from chapter6\b\kernel\kernel1.asm  
  
hwint00:      ; Interrupt routine for irq 0 (the clock).  
  
    mov al, EOI      ; '. reenable  
  
    out INT_M_CTL, al    ; / master 8259  
  
    iretd
```

代码段 6.8 时钟中断处理程序

```
; from chapter6\b\kernel\kernel5.asm  
  
ALIGN 16  
  
hwint00:      ; Interrupt routine for irq 0 (the clock).  
  
    sub esp, 4  
  
    pushad      ; '.  
  
    pushds      ; |  
  
    pushes      ; | 保存原寄存器值  
  
    pushfs      ; |  
  
    pushgs      ; /  
  
    mov dx, ss  
  
    mov ds, dx  
  
    mov es, dx  
  
  
  
    mov esp, StackTop      ; 切到内核栈  
  
    inc byte [gs:0]      ; 改变屏幕第 0 行, 第 0 列的字符
```

```

mov al,EOI                ;`.reenable

out INT_M_CTL,al          ;/ master 8259

mov esp,[p_proc_ready] ; 离开内核栈

lea  eax,[esp + P_STACKTOP]

mov dword [tss + TSS3_S_SP0],eax


pop  gs  ;`.
pop  fs  ; |
pop  es  ; | 恢复原寄存器值
pop  ds  ; |
popad    ;/

add esp,4

Iretld

```

6.2.2 代码主要结构

时钟中断处理程序：

- 保存原寄存器值
- 切到内核栈
- 设置 EOI
- 改变屏幕第 0 行，第 0 列的字符
- 离开内核栈
- 恢复原寄存器值
- 结束中断

6.2.3 调试过程

调试过程同 6.1.3，借助 Makefile 可以轻松地完成调试，只需要输入 **make**，即可由 GNU Make 自动完成编译链接工作，只需要输入 **bochs** 即可开始调试。调试结果如图 6.3 所示。

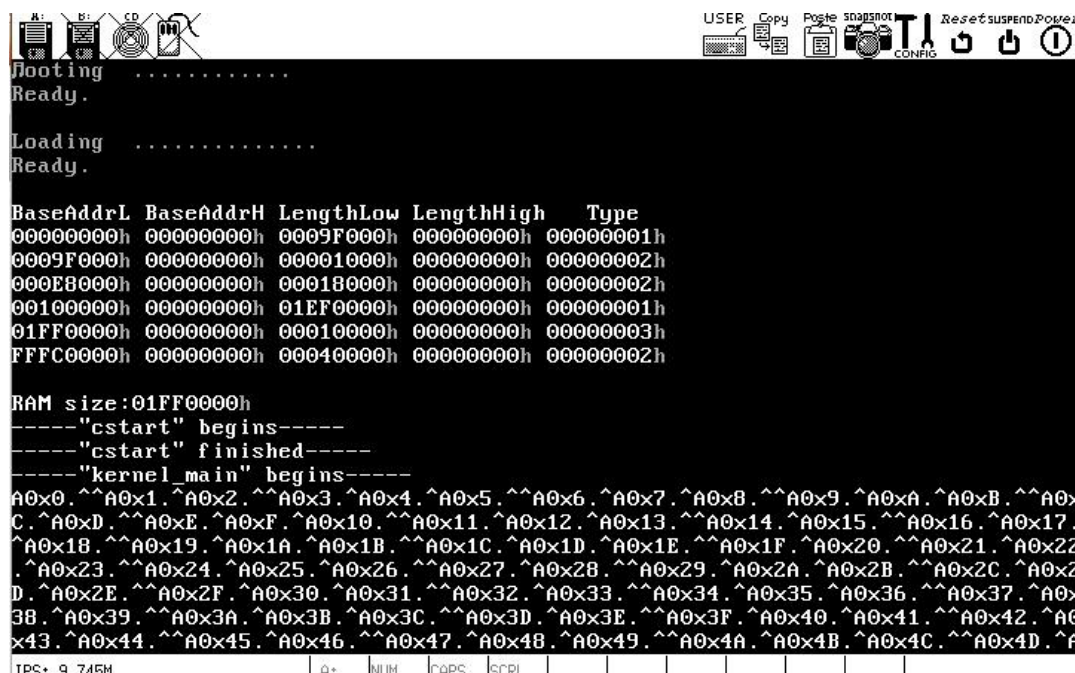


图 6.3 调试结果

从调试结果可以看到不断出现的字符“^”，这说明函数 `disp_str` 运行正常，而且没有影响到中断处理的其他部分以及进程 A。之所以在两次字符 A 的中间出现不止一个“^”，是因为笔者在进程的执行体中加入了 `delay()` 函数，而在此函数的执行过程中发生了多次中断。

6.2.4 实验分析

在本实验中，笔者主要完成了以下的内容：

第一，让时钟中断开始起作用，即通过 `out_byte` 向 8259A 芯片发信息，设置 `OCW1`，打开时钟中断。

第二,为了使时钟中断不停地发生而不是只发生一次,还需要通过指令设置FOI。

第三，进行现场的保护与恢复，即保存与恢复各个寄存器的值，通过 `push` 和 `pop` 指令来完成。

第四，则是为 `tss.esp0` 赋值，由于中断被打开，`cpu` 在 `ring0` 和 `ring1` 之间频

繁地切换，其中包括代码的切换和堆栈的切换，为了保证 ring1->ring0 可以正确返回，必须在使用 irted 返回前保证 tss.esp0 的值是正确的。

第五，则是实现了内堆栈，在调用函数时一定会用到堆栈，这样会破坏进程表，因此笔者需要切换到内堆栈，将 esp 指向其他位置，防止破坏数据。

6.3 中断重入

6.3.1 关键代码

代码段 6.9 时钟中断处理程序

```
; from chapter6\c\kernel\kernel.asm
hwint00:      ; Interrupt routine for irq 0 (the clock).

    sub esp, 4

    pushad     ; '\

    pushds    ; |

    pushes    ; | 保存原寄存器值

    pushfs    ; |

    pushgs    ; /

    mov dx, ss

    mov ds, dx

    mov es, dx

    inc byte [gs:0]      ; 改变屏幕第 0 行, 第 0 列的字符

    mov al, EOI          ; '\ reenable

    out INT_M_CTL, al    ; / master 8259

    inc dword [k_reenter]

    cmp dword [k_reenter], 0

    jne .re_enter

    mov esp, StackTop    ; 切到内核栈

    sti

    pushclock_int_msg
```



```

    call disp_str

    add esp, 4

;;; push 1
;;; call delay
;;; add esp, 4

cli

mov esp, [p_proc_ready] ; 离开内核栈

lea  eax, [esp + P_STACKTOP]

mov dword [tss + TSS3_S_SP0], eax

.re_enter: ; 如果(k_reenter != 0), 会跳转到这里

    dec dword [k_reenter]

    pop gs    ; `
    pop fs    ; |
    pop es    ; | 恢复原寄存器值
    pop ds    ; |
    popad     ; /

    add esp, 4

iretd

```

6.3.2 代码主要结构

时钟中断处理程序：

- 保存原寄存器值
- 改变屏幕第 0 行，第 0 列的字符
- 设置 EOI

得多。如果注释掉调用 `Delay` 的代码，可看到两者的速度又会一样了。

可以看到，字符 `A` 和相应的数字在不断出现，并且可以发现屏幕左上角的字母跳动速度快，而字符“^”打印速度慢，说明很多时候程序在执行了 `inc byte [gs:0]` 之后并没有执行 `disp_str`，这也说明中断重入的确发生了。


```

p_proc->ldt_sel = selector_ldt;

memcpy(&p_proc->ldts[0], &gdt[SELECTOR_KERNEL_CS >> 3],
      sizeof(DESCRIPTOR));
p_proc->ldts[0].attr1 = DA_C | PRIVILEGE_TASK << 5;
memcpy(&p_proc->ldts[1], &gdt[SELECTOR_KERNEL_DS >> 3],
      sizeof(DESCRIPTOR));
p_proc->ldts[1].attr1 = DA_DRW | PRIVILEGE_TASK << 5;
p_proc->regs.cs = ((8 * 0) & SA_RPL_MASK & SA_TI_MASK)
    | SA_TIL | RPL_TASK;
p_proc->regs.ds = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
    | SA_TIL | RPL_TASK;
p_proc->regs.es = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
    | SA_TIL | RPL_TASK;
p_proc->regs.fs = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
    | SA_TIL | RPL_TASK;
p_proc->regs.ss = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
    | SA_TIL | RPL_TASK;
p_proc->regs.gs = (SELECTOR_KERNEL_GS & SA_RPL_MASK)
    | RPL_TASK;

p_proc->regs.eip = (u32)p_task->initial_eip;
p_proc->regs.esp = (u32)p_task_stack;
p_proc->regs.eflags = 0x1202; /* IF=1, IOPL=1 */

p_task_stack -= p_task->stacksize;
p_proc++;
p_task++;
selector_ldt += 1 << 3;
}

proc_table[0].ticks = proc_table[0].priority = 24;
proc_table[1].ticks = proc_table[1].priority = 8;
proc_table[2].ticks = proc_table[2].priority = 6;
proc_table[3].ticks = proc_table[3].priority = 4;
proc_table[4].ticks = proc_table[4].priority = 3;

k_reenter = 0;
ticks = 0;

p_proc_ready = proc_table;

/* 初始化 8253 PIT */
out_byte(TIMER_MODE, RATE_GENERATOR);

```

```

        out_byte(TIMER0, (u8) (TIMER_FREQ/HZ) );
        out_byte(TIMER0, (u8) ((TIMER_FREQ/HZ) >> 8));

        put_irq_handler(CLOCK_IRQ, clock_handler); /* 设定时钟中断处理程序
*/
        enable_irq(CLOCK_IRQ);                      /* 让 8259A 可以接收时钟中
断 */

        restart();

        while(1){}
    }
; 此处省略 TestA、TestB、TestC 函数

void TestD()
{
    int i = 0x3000;
    while(1){
        disp_str("D.");
        milli_delay(10);
    }
}

void TestE()
{
    int i = 0x4000;
    while(1){
        disp_str("E.");
        milli_delay(10);
    }
}

```

代码段 6.11 global.c

```

; from chapter6\new\kernel\global.c
#define GLOBAL_VARIABLES_HERE

#include "type.h"
#include "const.h"
#include "protect.h"
#include "proto.h"
#include "proc.h"
#include "global.h"

PUBLIC PROCESS                proc_table[NR_TASKS];

```

```

PUBLIC char          task_stack[STACK_SIZE_TOTAL];

PUBLIC TASK   task_table[NR_TASKS] = {{TestA, STACK_SIZE_TESTA,
"TestA"},

          {TestB, STACK_SIZE_TESTB, "TestB"},
          {TestC, STACK_SIZE_TESTC, "TestC"},
          {TestD, STACK_SIZE_TESTD, "TestD"},
          {TestE, STACK_SIZE_TESTE, "TestE"},
          };

PUBLIC irq_handler    irq_table[NR_IRQ];

PUBLIC system_call    sys_call_table[NR_SYS_CALL] =
{sys_get_ticks};

```

代码段 6.12 proc.h

```

; from chapter6\new\include\proc.h
;省略 typedef 定义
/* Number of tasks */
#define NR_TASKS 5

/* stacks of tasks */
#define STACK_SIZE_TESTA 0x8000
#define STACK_SIZE_TESTB 0x8000
#define STACK_SIZE_TESTC 0x8000
#define STACK_SIZE_TESTD 0x8000
#define STACK_SIZE_TESTE 0x8000

#define STACK_SIZE_TOTAL (STACK_SIZE_TESTA + \
                          STACK_SIZE_TESTB + \
                          STACK_SIZE_TESTC + \
                          STACK_SIZE_TESTD + \
                          STACK_SIZE_TESTE)

```

代码段 6.13 proto.h

```

; from chapter6\new\include\proto.h
/* klib.asm */
PUBLIC void   out_byte(u16 port, u8 value);
PUBLIC u8   in_byte(u16 port);
PUBLIC void   disp_str(char * info);
PUBLIC void   disp_color_str(char * info, int color);
/* protect.c */
PUBLIC void   init_prot();

```


7 输入/输出系统

7.1 从中断开始——键盘初体验

7.1.1 关键代码

代码段 7.1 键盘中断处理程序

```
;from chapter7\a\kernel\keyboard.c

PUBLIC void keyboard_handler(int irq)

{

    disp_str("*");

}
```

代码段 7.2 打开键盘中断

```
;from chapter7\a\kernel\keyboard.c

PUBLIC void init_keyboard()

{

    put_irq_handler(KEYBOARD_IRQ, keyboard_handler);/*设定键盘中断处理程序*/

    enable_irq(KEYBOARD_IRQ);                      /*开键盘中断*/

}
```

代码段 7.3 调用 init_keyboard

```
;from chapter7\a\kernel\main.c

PUBLIC int kernel_main()

{

    ...

    init_keyboard();

    ...

}
```

7.1.2 代码主要结构

键盘中断流程：

- 调用键盘中断
- 打开键盘中断

- 接收键盘敲击后输出字符
- 结束调用

7.1.3 调试过程

先输入 **make image**，即可由 GNU Make 自动完成编译链接工作。再输入 **bochs** 即可开始调试，之后再输入 **c** 并按下回车，便可以进入调试界面。

调试结果如图 7.1 所示，可以看到，敲击键盘之后，程序中出现了一个“*”的字符；但再次敲击键盘后，键盘却不会再次响应，即屏幕上不会出现第二个“*”字符。

```
Booting ....., Ready.
Loading ....., Ready.
-----
BaseAddrL BaseAddrH LengthLow LengthHigh  Type
00000000h 00000000h 0009F000h 00000000h 00000001h
0009F000h 00000000h 00001000h 00000000h 00000002h
000E8000h 00000000h 00018000h 00000000h 00000002h
00100000h 00000000h 01EF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFC0000h 00000000h 00040000h 00000000h 00000002h

RAM size:01FF0000h
----"cstart" begins----
----"cstart" finished----
----"kernel_main" begins----
*
-----
```

图 7.1 在键盘中断处理程序打印星号

7.1.4 实验分析

在本实验中，笔者调用键盘中断，在敲击键盘后出现了一个“*”字符，代表键盘中断是准确无误的。但它目前只能响应一次键盘敲击，这显然是有问题的，需要进一步的改进。

7.2 从缓冲区读取信息并打印读取的值

7.2.1 关键代码

代码段 7.4 keyboard.c

```
;from chapter7\b\kernel\keyboard.c
PUBLIC void init_keyboard()
{
    put_irq_handler(KEYBOARD_IRQ, keyboard_handler);/*设定键盘中断处理程序*/
    enable_irq(KEYBOARD_IRQ);                      /*开键盘中断*/
}
```

7.2.2 代码主要结构

键盘中断流程:

- 调用键盘中断
- 打开键盘中断
- 接收键盘敲击送入缓冲区
- 读取键盘缓冲区并且打印返回值
- 关闭键盘中断
- 结束调用

7.2.3 调试过程

先输入 **make image**, 即可由 GNU Make 自动完成编译链接工作。再输入 **bochs** 即可开始调试, 之后再输入 **c** 并按下回车, 便可以进入调试界面。调试结果如下图所示。

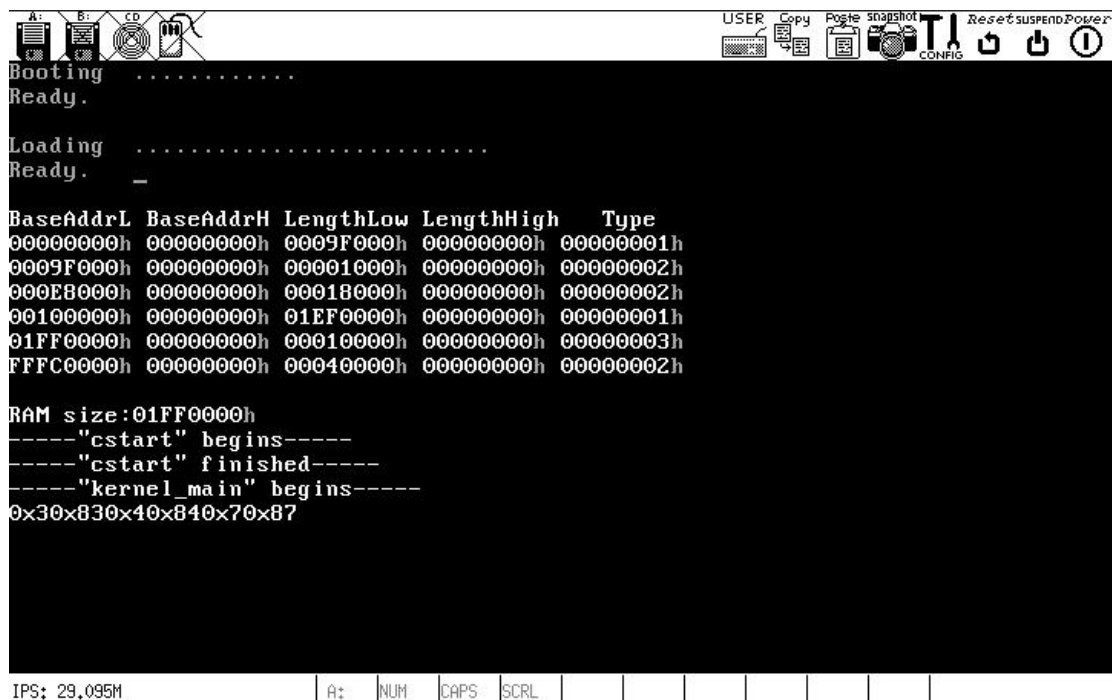


图 7.2 每发生一次键盘中断就读取一次 8042 缓冲区并打印读取的值

在上图中，笔者连续敲击了 3 个键：字符“2”、字符“3”和字符“6”，而实验结果则一共出现了 6 组代码：0x3、0x83，0x4、0x84，0x7、0x87，分别对应于字符“2”、字符“3”和字符“6”的 Make Code 和 Break Code。

7.2.4 实验分析

在本实验中，笔者对键盘中断的处理程序进行了改进，不再是简单的打印一个“*”，而是读取键盘缓冲区并且打印返回值，可以看到，目前的程序已经不会再卡死，而是可以响应多次键盘敲击过程，并且打印出扫描码。

对于这一点，则涉及键盘控制器 8042 芯片和键盘编码器 8048 芯片，它们会监视键盘的输入，并把适当的数据传给计算机。

而对于敲击键盘的动作，则会产生扫描码，分为按下一个按键或保持一个按键的 Make Code 和键盘弹起时的 Break Code，这些扫描码可以通过 in al,60h 读取，并且只有将扫描码从缓冲区中读出来后，8042 才能继续响应新的按键，这也解释了在实验 7.1 中只打印一个“*”的原因。

7.3 扫描码解析数组、键盘输入缓冲区、与使用新加的任务处理键盘操作

7.3.1 关键代码

代码段 7.5 键盘缓冲区

```
;from chapter7\c\include\keyboard.h
/* Keyboard structure, 1 per console. */
typedef struct s_kb {
    char*    p_head;        /* 指向缓冲区中下一个空闲位置 */
    char*    p_tail;        /* 指向键盘任务应处理的字节 */
    int  count;             /* 缓冲区中共有多少字节 */
    char buf[KB_IN_BYTES]; /* 缓冲区 */
}KB_INPUT;
```

代码段 7.6 修改后的 keyboard_handler

```
;from chapter7\c\kernel\keyboard.c
PUBLIC void keyboard_handler(int irq)
{
    u8 scan_code = in_byte(KB_DATA);

    if (kb_in.count < KB_IN_BYTES) {
        *(kb_in.p_head) = scan_code;
        kb_in.p_head++;
        if (kb_in.p_head == kb_in.buf + KB_IN_BYTES) {
            kb_in.p_head = kb_in.buf;
        }
        kb_in.count++;
    }
}
```

代码段 7.7 修改后的 init_keyboard

```
;from chapter7\c\kernel\keyboard.c
```

```

PUBLIC void init_keyboard()
{
    kb_in.count = 0;

    kb_in.p_head = kb_in.p_tail = kb_in.buf;

    put_irq_handler(KEYBOARD_IRQ, keyboard_handler);/*设定键盘中断处理程序*/

    enable_irq(KEYBOARD_IRQ);                      /*开键盘中断*/
}

```

代码段 7.8 关于键盘读取的 tty 任务

```

;from chapter7\c\kernel\tty.c

PUBLIC void task_tty()
{
    while (1) {
        keyboard_read();
    }
}

```

代码段 7.9 键盘读取函数 keyboard_read()

```

;from chapter7\c\kernel\keyboard.c

PUBLIC void keyboard_read()
{
    u8 scan_code;

    if(kb_in.count > 0){
        disable_int();

        scan_code = *(kb_in.p_tail);

        kb_in.p_tail++;

        if (kb_in.p_tail == kb_in.buf + KB_IN_BYTES) {
            kb_in.p_tail = kb_in.buf;
        }

        kb_in.count--;
    }
}

```

```

        enable_int();

        disp_int(scan_code);

    }

}

```

7.3.2 代码主要结构

键盘中断流程：

- 调用键盘中断
- 打开键盘中断
- 读取键盘输入送入缓冲区
- 读取缓冲区字符
- 通过任务 tty 处理扫描码
- 打印返回值
- 结束调用

7.3.3 调试过程

先输入 **make image**, 即可由 GNU Make 自动完成编译链接工作。再输入 **bochs** 即可开始调试, 之后再输入 **c** 并按下回车, 便可以进入调试界面。调试结果如图 7.3 所示。

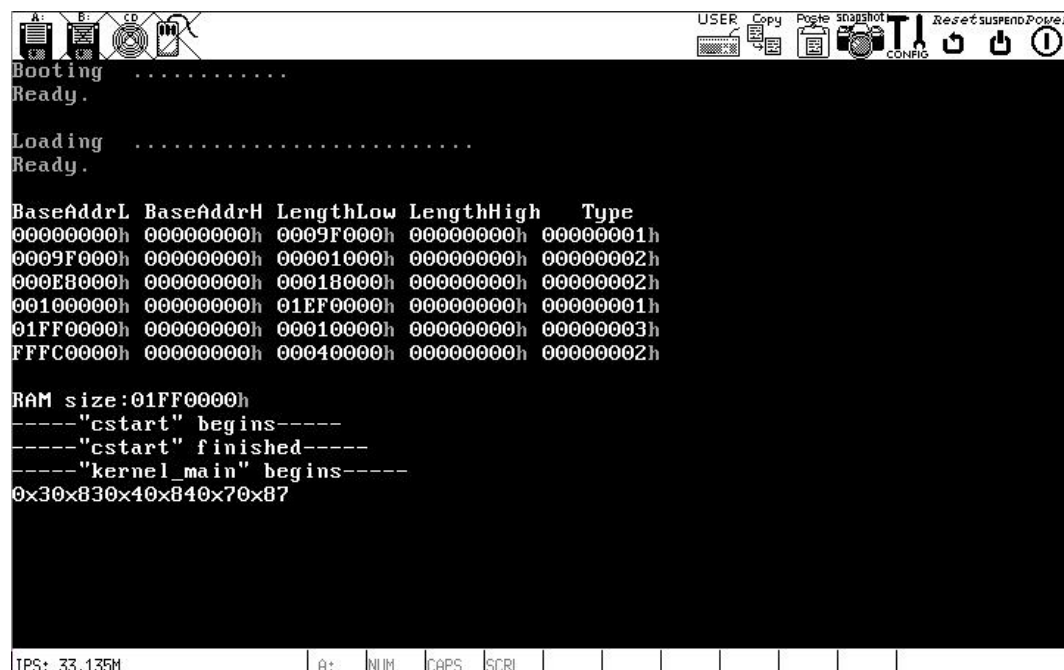


图 7.3 调试结果

可以看到，调试结果同前一个实验相比并没有出现变化，因为笔者仅仅只是通过任务来处理扫描码，但还没有对扫描码进行解析。

7.3.4 实验分析

在本实验中，笔者首先是建立了一个扫描码的解析数组，其次则是建立了一个结构体，用来承担键盘缓冲区的工作，在最后，则是添加了一个任务，用来处理键盘操作，而这个任务将会构成未来终端任务的一部分。

7.4 解析扫描码——让字符显示出来

7.4.1 关键代码

代码段 7.10 keyboard.c

```
;from chapter7\d\kernel\keyboard.c

PUBLIC void keyboard_read()
{
    u8  scan_code;

    char output[2];

    int  make;    /* TRUE: make;  FALSE: break. */

    memset(output, 0, 2);

    if(kb_in.count > 0){
        disable_int();

        scan_code = *(kb_in.p_tail);

        kb_in.p_tail++;

        if (kb_in.p_tail == kb_in.buf + KB_IN_BYTES) {
            kb_in.p_tail = kb_in.buf;
        }

        kb_in.count--;

        enable_int();

        /* 下面开始解析扫描码 */

        if (scan_code == 0xE1) {
            /* 暂时不做任何操作 */
        }

        else if (scan_code == 0xE0) {
            /* 暂时不做任何操作 */
        }
    }
}
```

```

else {    /* 下面处理可打印字符 */

    /* 首先判断 Make Code 还是 Break Code */

    make = (scan_code & FLAG_BREAK ? FALSE : TRUE);

    /* 如果是 Make Code 就打印，是 Break Code 则不做处理 */

    if(make) {

        output[0] = keymap[(scan_code&0x7F)*MAP_COLS];

        disp_str(output);

    }

    /* disp_int(scan_code); */

}
}
}

```

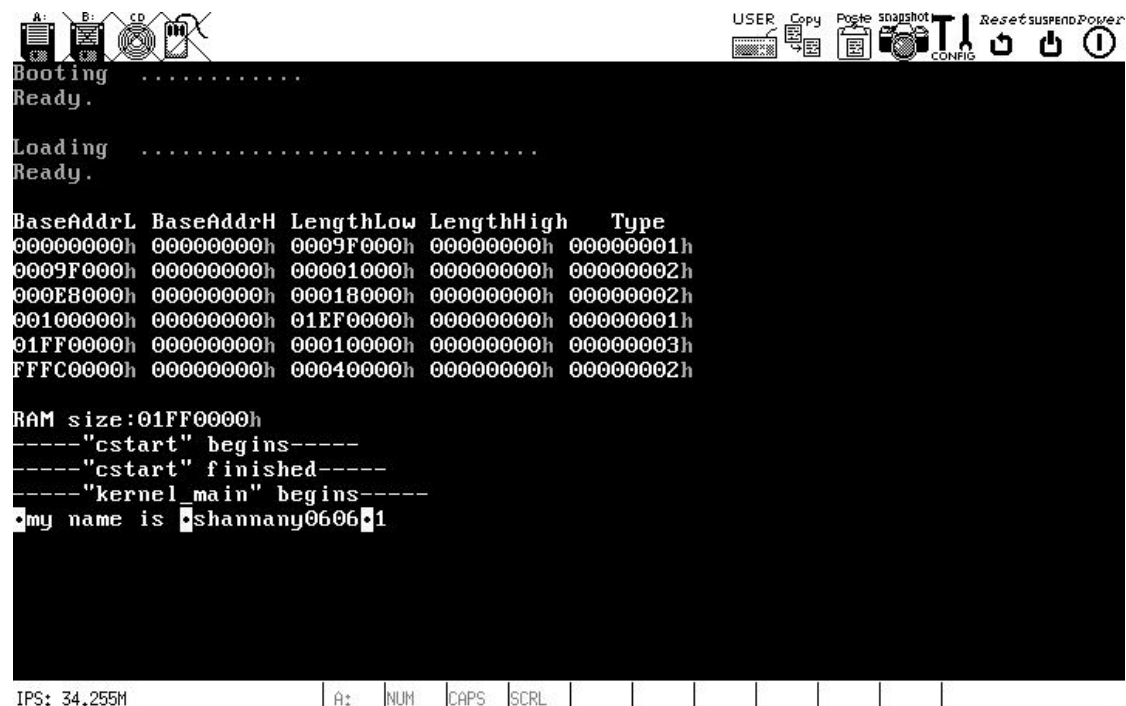
7.4.2 代码主要结构

键盘中断流程：

- 调用键盘中断
- 打开键盘中断
- 读取键盘输入送入缓冲区
- 读取缓冲区字符
- 通过任务 tty 处理扫描码
- 扫描解析数组
- 打印 Make Code 的解析码
- 结束调用

7.4.3 调试过程

先输入 **make image**，即可由 GNU Make 自动完成编译链接工作。再输入 **bochs** 即可开始调试，之后再输入 **c** 并按下回车，便可以进入调试界面。调试结果如图 7.4 所示。



The screenshot shows a debugger window with a black background and white text. At the top, there are icons for A:, B:, CD, and a power button. On the right, there are icons for USER, Copy, Paste, snapshot, CONFIG, Reset, suspend, and Power. The main text area displays the following information:

```
Booting .....  
Ready.  
  
Loading .....  
Ready.  
  
BaseAddrL BaseAddrH LengthLow LengthHigh Type  
00000000h 00000000h 0009F000h 00000000h 00000001h  
0009F000h 00000000h 00001000h 00000000h 00000002h  
000E8000h 00000000h 00018000h 00000000h 00000002h  
00100000h 00000000h 01EF0000h 00000000h 00000001h  
01FF0000h 00000000h 00010000h 00000000h 00000003h  
FFFC0000h 00000000h 00040000h 00000000h 00000002h  
  
RAM size:01FF0000h  
-----"cstart" begins-----  
-----"cstart" finished-----  
-----"kernel_main" begins-----  
•my name is •shannany0606!•1
```

At the bottom, there is a status bar with the text "IPS: 34.255M" and a row of icons: A:, NUM, CAPS, SCRL, and several empty boxes.

图 7.4 调试结果

笔者输入的字符串实际上是“My name is Shannany0606!”。根据打印结果与输入的差异可知，目前程序已经可以对小写 a-z 以及 0-9 进行正常的响应，但仍然无法打印出大写字母，对 shift、alt 和 ctrl 等按键则会输出意义不明的字符。

7.4.4 实验分析

在本实验中笔者通过扫描码解析数组 `keymap[]` 对扫描码进行了初步的解析，可以对输入的简单字符和数字进行解析，但仍然无法处理组合键以及特殊字符，这也是之后的实验需要解决的问题。

7.5 功能改进

对本书配套的第 7 章最终版代码，按下 TAB 键后的运行结果如下：

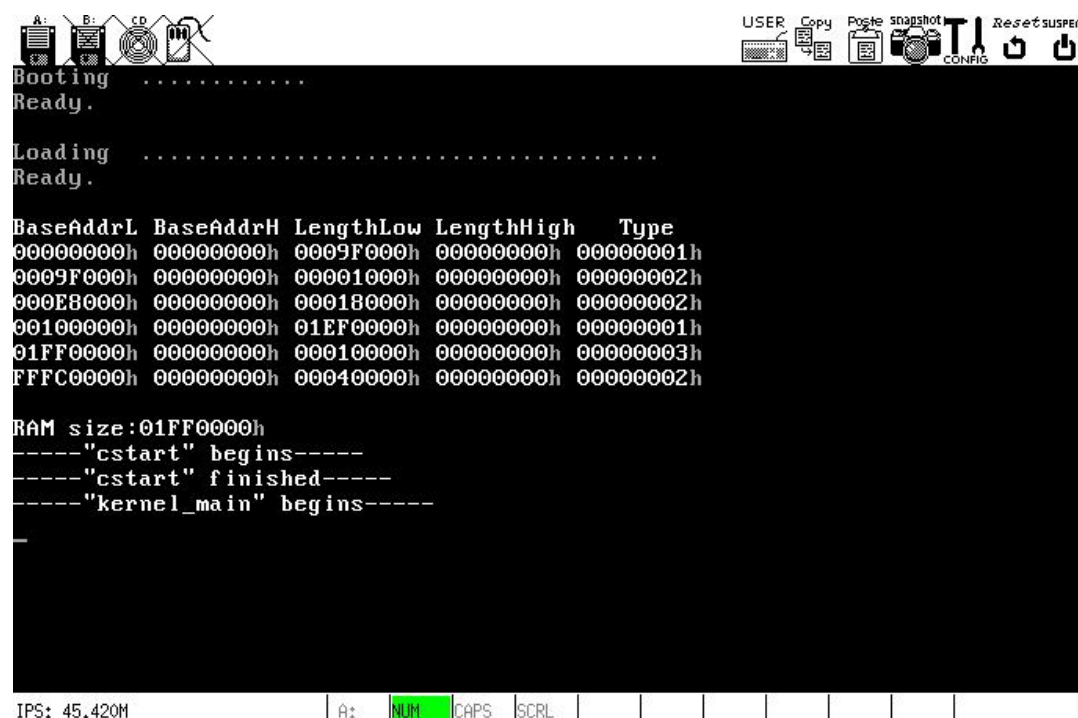


图 7.5 改进前的调试结果

可以发现，本书配套代码对 TAB 键输入没有响应。对这一结果，笔者通过在文件 keyboard.c 和 tty.c 中增添对 TAB 键的响应，使其输出两个空格。具体补充内容如代码段 7.11、7.12 中标红字段所示。

代码段 7.11 keyboard.c

```
; from chapter7\new\kernel\keyboard.c
; 这一文件中仅修改了 keyboard_read 函数
PUBLIC void keyboard_read(TTY* p_tty) // 公共函数，读取键盘输入，参数为终端设备结构体的指针
{
    u8 scan_code; // 定义一个无符号 8 位整数变量用于存放键盘扫描码
    char output[2]; // 定义一个字符数组，用于保存输出字符
    int make; // 定义一个整数变量，用于标识按键是按下（1）还是释放（0）
    u32 key = 0; // 定义一个无符号 32 位整数变量，用于存放正在处理的键值，例如 Home 键被按下/释放，则 key 值为 'HOME'
    u32* keyrow; // 定义一个无符号 32 位整数指针，用于指向 keymap 数组的一行

    if(kb_in.count > 0) { // 判断键盘输入缓冲区的计数值是否大于 0，如果大于 0，说明有按键输入
```

```

        code_with_E0 = 0; // 重置 code_with_E0 变量，用于判断是否有 0xE0 前缀的
        扫描码

        scan_code = get_byte_from_kbuf(); // 从键盘缓冲区读取一个字节到
        scan_code

        /* 下面开始解析扫描码 */
        if (scan_code == 0xE1) { // 如果扫描码等于 0xE1，说明可能按下了
        PauseBreak 键
            int i;
            u8 pausebrk_scode[] = {0xE1, 0x1D, 0x45,
                                   0xE1, 0x9D, 0xC5}; // PauseBreak 键的扫描码序列
            int is_pausebreak = 1;
            for(i=1;i<6;i++){
                if (get_byte_from_kbuf() != pausebrk_scode[i]) { // 遍历比
                较后续的扫描码是否与 PauseBreak 键的扫描码匹配
                    is_pausebreak = 0; // 如果不匹配，说明不是 PauseBreak 键，
                退出循环
                    break;
                }
            }
            if (is_pausebreak) { // 如果 is_pausebreak 为 1，说明按下了
            PauseBreak 键，key 值设置为 PAUSEBREAK
                key = PAUSEBREAK;
            }
        }
        else if (scan_code == 0xE0) { // 如果扫描码等于 0xE0，说明按下的可能是
        特殊键（如 PrintScreen）
            scan_code = get_byte_from_kbuf(); // 读取下一个字节的扫描码

            /* PrintScreen 被按下 */
            if (scan_code == 0x2A) {
                if (get_byte_from_kbuf() == 0xE0) {
                    if (get_byte_from_kbuf() == 0x37) {
                        key = PRINTSCREEN; // PrintScreen 键被按下
                        make = 1; // 标记按键状态为按下
                    }
                }
            }
            /* PrintScreen 被释放 */
            if (scan_code == 0xB7) {
                if (get_byte_from_kbuf() == 0xE0) {
                    if (get_byte_from_kbuf() == 0xAA) {
                        key = PRINTSCREEN; // PrintScreen 键被释放
                    }
                }
            }
        }
    }
}

```

```

        make = 0; // 标记按键状态为释放
    }
}

/* 不是 PrintScreen, 此时 scan_code 为 0xE0 紧跟的那个值. */
if (key == 0) {
    code_with_E0 = 1; // 设置 code_with_E0 标志, 表示有 0xE0 前缀的
扫描码
}
}

if ((key != PAUSEBREAK) && (key != PRINTSCREEN)) { // 如果按下的键
不是 PauseBreak 键和 PrintScreen 键
    make = (scan_code & FLAG_BREAK ? 0 : 1); // 确定按键的状态 (按
下或释放)

    // 定位到 keymap 数组中的一行
    keyrow = &keymap[(scan_code & 0x7F) * MAP_COLS];

    column = 0;

    int caps = shift_l || shift_r; // 判断是否按下了 Shift 键
    if (caps_lock) { // 判断是否按下了 CapsLock 键
        if ((keyrow[0] >= 'a') && (keyrow[0] <= 'z')) {
            caps = !caps; // 如果 CapsLock 被按下, 且当前键是字母键, 取
反 caps
        }
    }
    if (caps) { // 如果 caps 为真, 表示需要切换大小写, 将 column 设为 1
        column = 1;
    }

    if (code_with_E0) { // 如果 code_with_E0 为真, 表示有 0xE0 前缀的扫
描码, 将 column 设为 2
        column = 2;
    }

    key = keyrow[column]; // 从 keymap 中取出 key 值

    // 处理 Shift、Ctrl、Alt 等特殊按键的情况
    switch(key) {
    case SHIFT_L:
        shift_l = make; // 更新左 Shift 键的状态
        break;
    case SHIFT_R:

```

```

        shift_r = make; // 更新右 Shift 键的状态
        break;
    case CTRL_L:
        ctrl_l = make; // 更新左 Ctrl 键的状态
        break;
    case CTRL_R:
        ctrl_r = make; // 更新右 Ctrl 键的状态
        break;
    case ALT_L:
        alt_l = make; // 更新左 Alt 键的状态
        break;
    case ALT_R:
        alt_r = make; // 更新右 Alt 键的状态
        break;
    case CAPS_LOCK:
        if (make) {
            caps_lock = !caps_lock; // 更新 CapsLock 键的状态
            set_leds(); // 更新 LED 灯状态
        }
        break;
    case NUM_LOCK:
        if (make) {
            num_lock = !num_lock; // 更新 NumLock 键的状态
            set_leds(); // 更新 LED 灯状态
        }
        break;
    case SCROLL_LOCK:
        if (make) {
            scroll_lock = !scroll_lock; // 更新 ScrollLock 键的状态
            set_leds(); // 更新 LED 灯状态
        }
        break;
    default:
        break;
}

if (make) { // 如果是按下状态，则处理按键事件
    int pad = 0;

    /* 首先处理小键盘 */
    if ((key >= PAD_SLASH) && (key <= PAD_9)) {
        pad = 1; // 标记 pad 为 1，表示当前键在小键盘上
        switch(key) {
            case PAD_SLASH:

```

是数字字符

```
key = '/';// 小键盘上的除号对应的是 '/'
break;
case PAD_STAR:
key = '*';// 小键盘上的乘号对应的是 '*'
break;
case PAD_MINUS:
key = '-';// 小键盘上的减号对应的是 '-'
break;
case PAD_PLUS:
key = '+';// 小键盘上的加号对应的是 '+'
break;
case PAD_ENTER:
key = ENTER;
break;
default:
if (num_lock &&
    (key >= PAD_0) &&
    (key <= PAD_9)) {
key = key - PAD_0 + '0';// 小键盘上的数字键对应的
}
else if (num_lock &&
    (key == PAD_DOT)) {
key = '.';// 小键盘上的小数点键对应的是 '.'
}
else{
switch(key) {
case PAD_HOME:
key = HOME;
break;
case PAD_END:
key = END;
break;
case PAD_PAGEUP:
key = PAGEUP;
break;
case PAD_PAGEDOWN:
key = PAGEDOWN;
break;
case PAD_INS:
key = INSERT;
break;
case PAD_UP:
key = UP;
```



```

        break;
    case PAD_DOWN:
        key = DOWN;
        break;
    case PAD_LEFT:
        key = LEFT;
        break;
    case PAD_RIGHT:
        key = RIGHT;
        break;
    case PAD_DOT:
        key = DELETE;
        break;
    case TAB:
        key = TAB;
        break;
    default:
        break;
    }
}
break;
}
}
key |= shift_l ? FLAG_SHIFT_L : 0; // 如果左 Shift 键被
按下, 综合考虑该情况
key |= shift_r ? FLAG_SHIFT_R : 0; // 如果右 Shift 键被
按下, 综合考虑该情况
key |= ctrl_l ? FLAG_CTRL_L : 0; // 如果左 Ctrl 键被按下, 综
合考虑该情况
key |= ctrl_r ? FLAG_CTRL_R : 0; // 如果右 Ctrl 键被按下, 综
合考虑该情况
key |= alt_l ? FLAG_ALT_L : 0; // 如果左 Alt 键被按下, 综合
考虑该情况
key |= alt_r ? FLAG_ALT_R : 0; // 如果右 Alt 键被按下, 综合
考虑该情况
key |= pad ? FLAG_PAD : 0; // 如果小键盘的键被按下,
综合考虑该情况

in_process(p_tty, key); // 调用 in_process 函数, 对按键进行处
理
}
}
}
}

```

代码段 7.12 tty.c

```
; from chapter7\new\kernel\tty.c
; 这一文件中仅修改了 in_process 函数
PUBLIC void in_process(TTY* p_tty, u32 key)
{
    char output[2] = {'\0', '\0'};

    if (!(key & FLAG_EXT)) {
        put_key(p_tty, key);
    }
    else {
        int raw_code = key & MASK_RAW;
        switch(raw_code) {
            case ENTER:
                put_key(p_tty, '\n');
                break;
            case BACKSPACE:
                put_key(p_tty, '\b');
                break;
            case UP:
                if ((key & FLAG_SHIFT_L) || (key & FLAG_SHIFT_R)) {
                    scroll_screen(p_tty->p_console, SCR_DN);
                }
                break;
            case DOWN:
                if ((key & FLAG_SHIFT_L) || (key & FLAG_SHIFT_R)) {
                    scroll_screen(p_tty->p_console, SCR_UP);
                }
                break;
            case TAB:
                put_key(p_tty, ' ');
                put_key(p_tty, ' ');
                break;
            case F1:
            case F2:
            case F3:
            case F4:
            case F5:
            case F6:
            case F7:
            case F8:
            case F9:
            case F10:
```

```

case F11:
case F12:
    /* Alt + F1~F12 */
    if ((key & FLAG_ALT_L) || (key & FLAG_ALT_R)) {
        select_console(raw_code - F1);
    }
    break;
    default:
        break;
    }
}
}

```

功能改进后，代码输出结果如图 7.6 所示。可以看到，对于 TAB 键输入，代码输出了两个空格，这表明笔者对程序的功能改进成功达成既定目标。

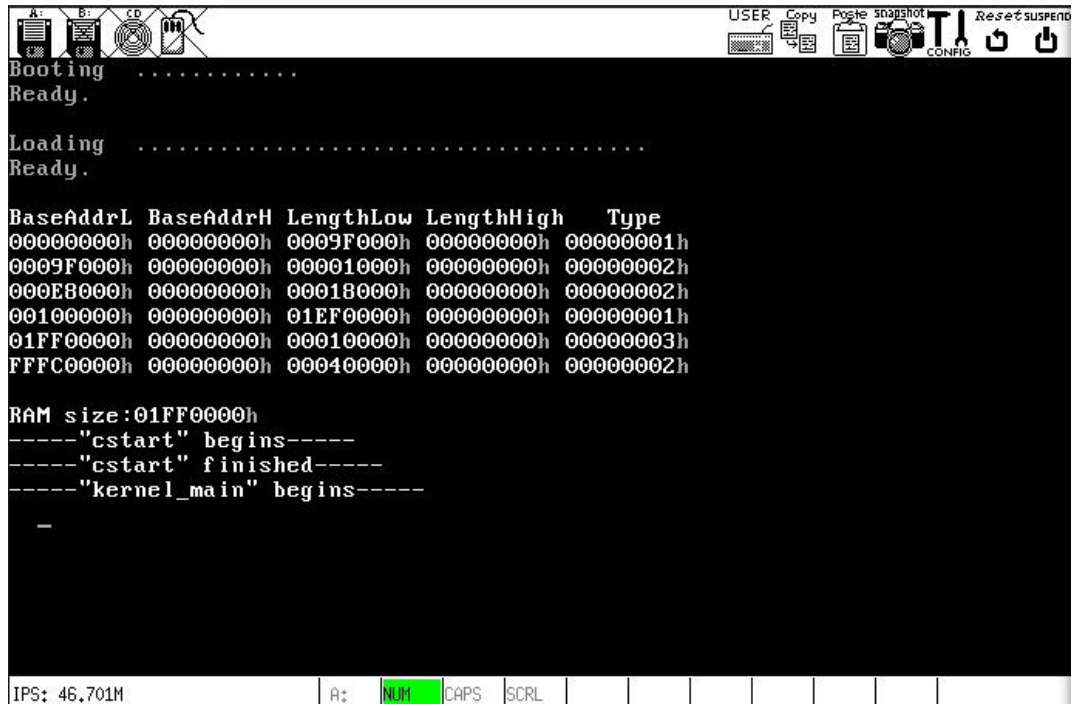


图 7.6 改进后的调试结果

8 实验总结

8.1 实验中经常遇到的错误

8.1.1 bochs 配置文件 bochsrc 有关的问题

对于书本配套的 bochsrc 配置文件，由于 bochs 版本不同，部分旧版本下能正确执行的语句在新版本下可能报错，故需进行修改。修改后的 bochsrc 文件如下所示：

代码段 8.1 bochsrc

```
#####  
# Configuration file for Bochs  
#####  
  
# how much memory the emulated machine will have  
megs: 32  
  
# filename of ROM images  
romimage: file=/usr/share/bochs/BIOS-bochs-latest  
# vgaromimage: /usr/share/vgabios/vgabios.bin  
vgaromimage: file=/usr/share/vgabios/vgabios.bin  
  
# what disk images will be used  
floppya: 1_44=a.img, status=inserted  
  
# choose the boot disk.  
boot: floppy  
  
# where do we send log messages?  
# log: bochsout.txt  
  
# disable the mouse
```

```
mouse: enabled=0

# enable key mapping, using US layout as default.
#keyboard_mapping:enabled=1,map=/usr/share/bochs/keymaps/x11-pc-us.map
keyboard: keymap=/usr/share/bochs/keymaps/x11-pc-us.map
```

总之，需要在 `vgaromimage` 和 `keyboard` 处进行修改，即上述代码中的标红部分。改正后的代码均在错误代码的下一行。

如此修改后，即可顺利地进行调试，否则可能出现以下错误提示：

- `keyboard_mapping` is deprecated - use 'keyboard' option instead
- `keyboard` directive malformed
- `vgaromimage` directive malformed

8.1.2 出现挂载点/mnt/floppy 不存在的提示

该问题出现的原因是 `/mnt` 目录下没有对应的 `/floppy` 文件夹。

解决办法是在 `/mnt` 目录下建立一个 `floppy` 文件夹，需要输入以下指令：

```
sudo mkdir /mnt/floppy
```

8.1.3 ld 指令出现 incompatible with i386:x86-64 output

在实验 5.1 中，输入 `ld -s hello.o -o hello` 会报错，这是由于安装的 Ubuntu 是 64 位，默认产生 64 位的目标代码，但此处应编译 32 位目标代码，所以 `ld` 连接指令应改为 `ld -m elf_i386 -s -o hello hello.o`，操作成功。

8.1.4 gcc 指令相关问题

实验 5.2 中教材上给出的参考指令为：

```
gcc -c -o bar.o bar.c
```

实际上，由于笔者安装的是 64 位 Linux 操作系统，编译产生的文件为 64 位代码，而不是教材中默认的 32 位代码。因此，需要根据环境变化对指令做出一定的修改：

```
gcc -m32 -c -o bar.o bar.c
```

8.1.5 调试过程中出现乱码、红色错误、输出“error”等问题

第六、七章教材配套代码是存在问题的，如果直接 **make image**，调试结果会出现图 8.1 所示的错误。

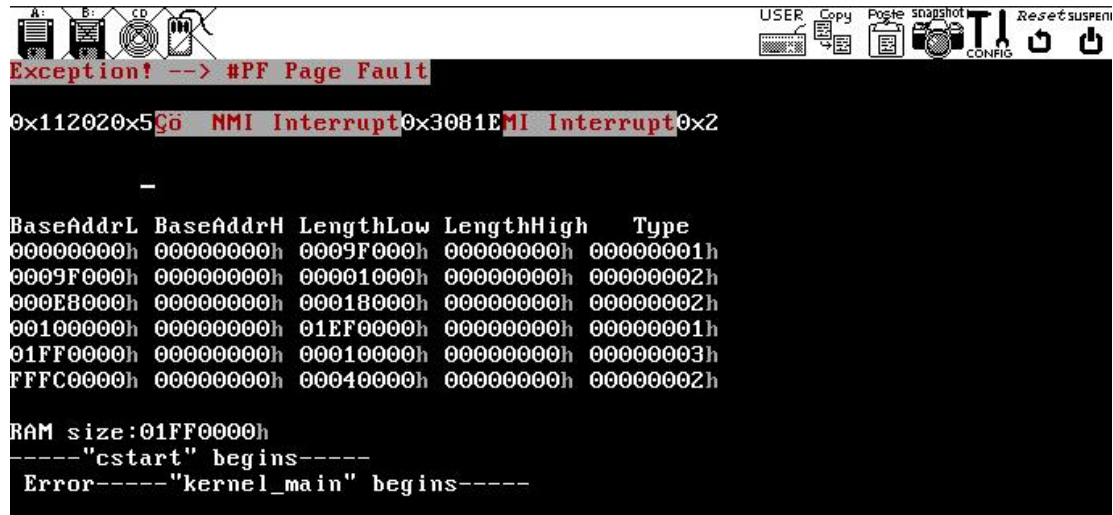


图 8.1 错误调试结果

实际上，问题出在代码 `lib/kliba.asm` 的 `disp_str` 函数中存在部分代码缺失，故函数 `disp_str` 应作如下修改（标红语句为新增代码）：

代码段 8.2 kliba.asm

```
; from chapter6\lib\kliba.asm

disp_str:

    push ebp
    mov ebp, esp

    push ebx
    push esi
    push edi

    mov esi, [ebp + 8] ; pszInfo
    mov edi, [disp_pos]
    mov ah, 0Fh

.L:
    lodsb
    test al, al
    jz .2
```

```

    cmp al, 0Ah ; 是回车吗?
    jnz .3
    pusheax
    mov eax, edi
    mov bl, 160
    div bl
    and eax, 0FFh
    inc eax
    mov bl, 160
    mul bl
    mov edi, eax
    pop eax
    jmp .1
.3:
    mov [gs:edi], ax
    add edi, 2
    jmp .1
.2:
    mov [disp_pos], edi
    pop edi
    pop esi
    pop ebx
    pop ebp
    ret

```

完成代码补全后，重新 **make image** 并调试，即可得到无报错的调试结果，如图 8.2 所示。

```

Booting .....
Ready.

Loading .....
Ready.

BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0009F000h 00000000h 00000001h
0009F000h 00000000h 00001000h 00000000h 00000002h
000E8000h 00000000h 00018000h 00000000h 00000002h
00100000h 00000000h 01EF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFC0000h 00000000h 00040000h 00000000h 00000002h

RAM size: 01FF0000h
----"cstart" begins----
----"cstart" finished----
----"kernel_main" begins----
A0x0.A0x1.A0x2.A0x3.A0x4.A0x5.A0x6.A0x7.A0x8.A0x9.A0xA.A0xB.A0xC.A0xD.A0xE.A0xF.
A0x10.A0x11.A0x12.A0x13.A0x14.A0x15.A0x16.A0x17.A0x18.A0x19.A0x1A.A0x1B.A0x1C.A0
x1D.A0x1E.A0x1F.A0x20.A0x21.A0x22.A0x23.A0x24.A0x25.A0x26.A0x27.A0x28.A0x29.A0x2
A.A0x2B.A0x2C.A0x2D.A0x2E.A0x2F.A0x30.A0x31.A0x32.A0x33.A0x34.A0x35.A0x36.A0x37.
A0x38.A0x39.A0x3A.A0x3B.A0x3C.A0x3D.A0x3E.A0x3F.A0x40.A0x41.A0x42.A0x43.A0x44.A0
x45.A0x46.A0x47.A0x48.A0x49.A0x4A.A0x4B.A0x4C.A0x4D.A0x4E.A0x4F.A0x50.A0x51.A0x5
2.A0x53.A0x54.A0x55.A0x56.A0x57.A0x58.A0x59.A0x5A.A0x5B.A0x5C.A0x5D.A0x5E.A0x5F.

IPS: 8,841M

```

图 8.2 无报错的调试结果

8.2 实验总结与心得

8.2.1 实验总结

在本实验中，笔者遵循《ORANGE’S：一个操作系统的实现》一书，进行了操作系统实验设计与实践的的实现，并且完成了该书前七章每章中的 3 个小实验。

在第一章的实验中，笔者通过汇编语言编写了一个“最简单”的操作系统，又或者说是个最简单的引导扇区，它可以在裸机上运行，并打印出红色的“Hello, OS World!”，如图 8.3 所示。

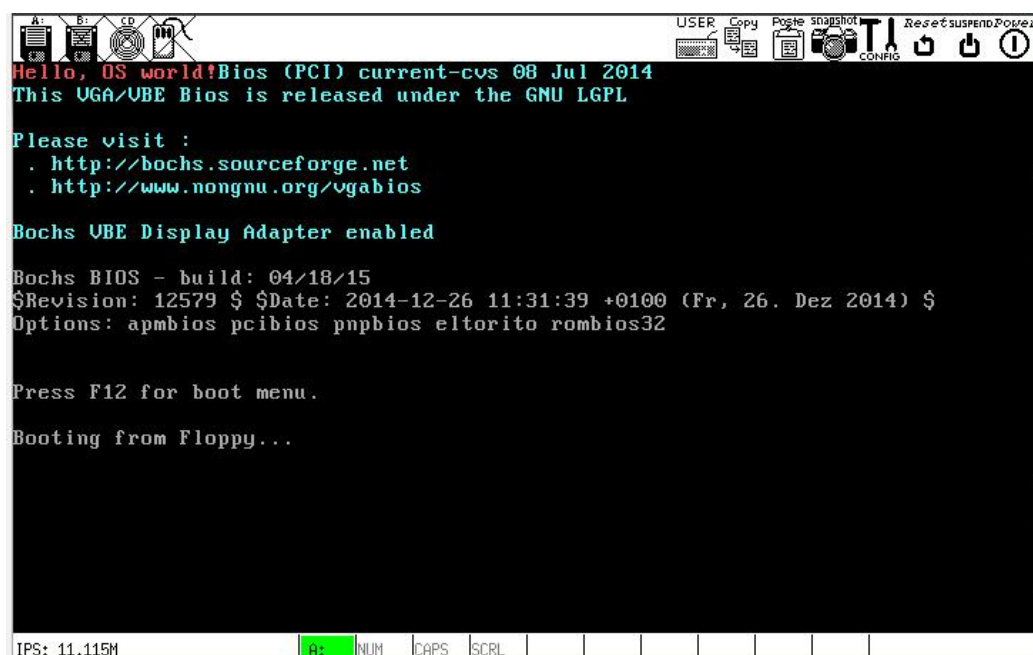


图 8.3 最小的“操作系统”

在第二章的实验中，笔者进行了实验环境的搭配，安装了虚拟机软件 VMware Workstation Pro，并且通过该软件在 Windows 电脑上运行了 Linux 操作系统中的 Ubuntu。之后，还编译了 Bochs 软件，安装了 GCC 等软件，对 Linux 操作系统有了一个初步的了解，虚拟机如图 8.4 所示。

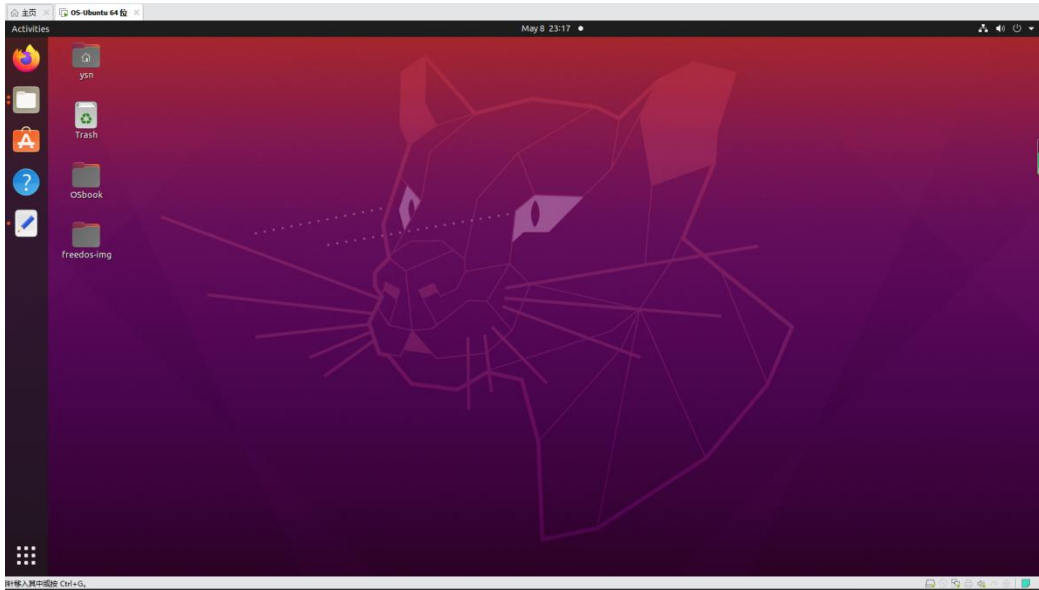


图 8.4 搭建工作环境——Ubuntu 虚拟机

在第三章的实验中，笔者主要学习了有关操作系统保护模式的相关知识，并且学习了有关 GDT、LDT、特权级、堆栈等的知识，除此之外，这章实验也通过汇编代码，完成了 CPU 从实模式进入保护模式以及从保护模式返回实模式的操作。这一章中出现了大量的汇编代码，还涉及很多具体的知识，难度较大，实验结果如图 8.5、图 8.6 和图 8.7 所示。

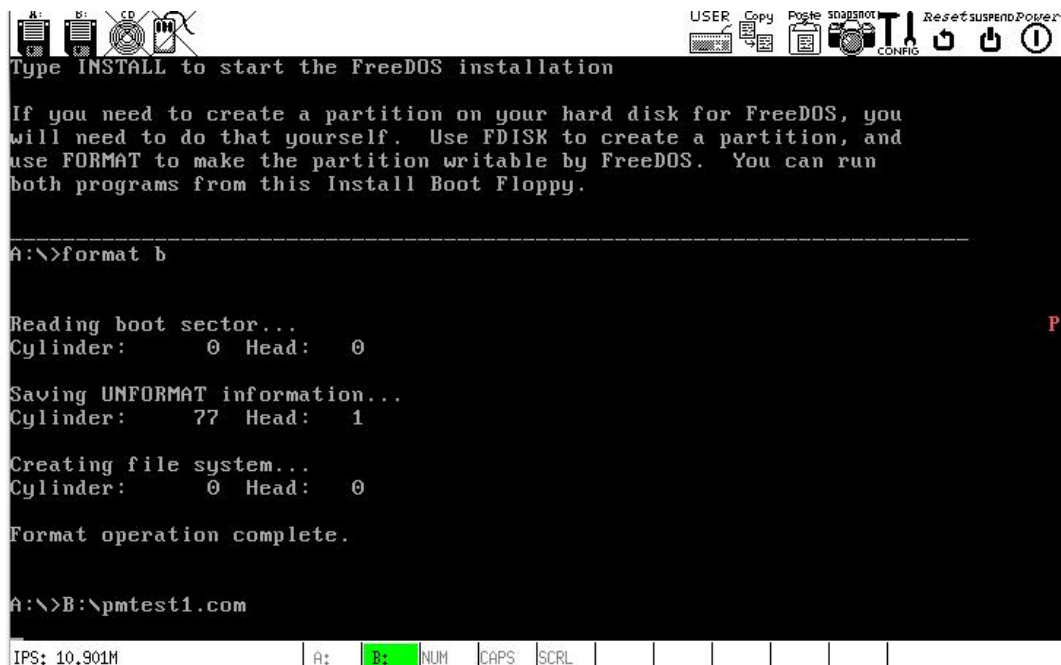


图 8.5 进入保护模式

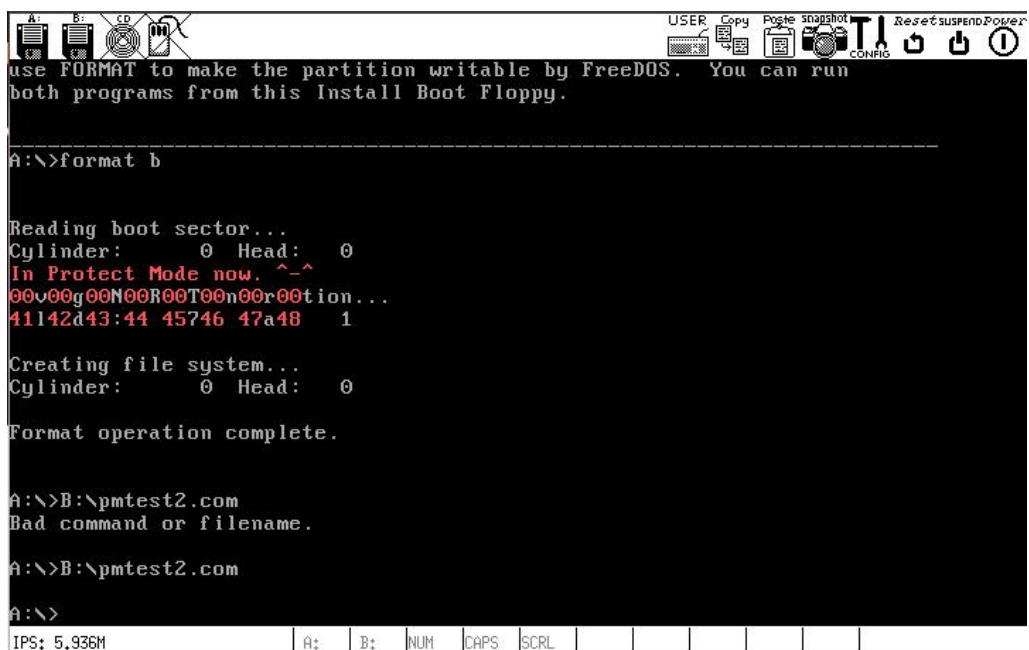


图 8.6 测试保护模式



图 8.7 返回实模式

在前三章的实验中，笔者或是运用系统的引导盘，或是借用 DOS 的引导盘，而在第四章的实验中，自己实现引导扇区 Boot，并且向程序加载器 Loader 交出了控制权，这时，操作系统已经不需要借助系统或者 DOS 才能运行了，甚至只要一个.COM 文件中不含有系统调用，就都可以通过“操作系统”运行它，实验结果如图 8.8 所示。

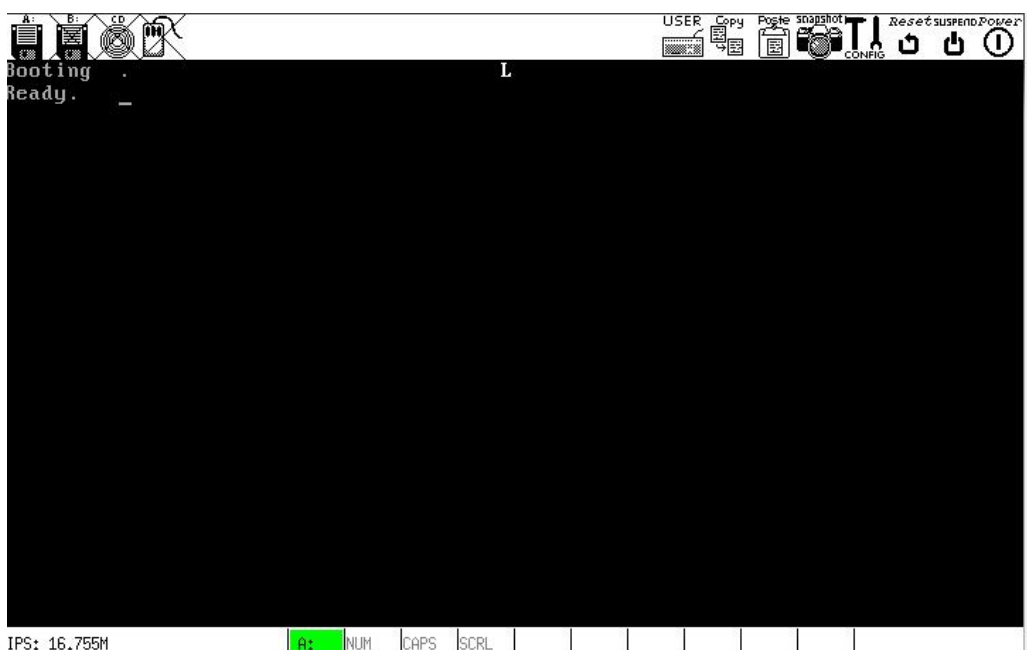


图 8.8 自制的 Loader

在第五章的实验中，笔者则开始编写操作系统的内核雏形。在第四章的实验中，笔者已经自制了引导磁盘 Boot 和 Loader，并且将控制权移交给了 Loader，而在第五章的实验中，则是由 Loader 进一步地载入内核 Kernel，如图 8.9 所示。

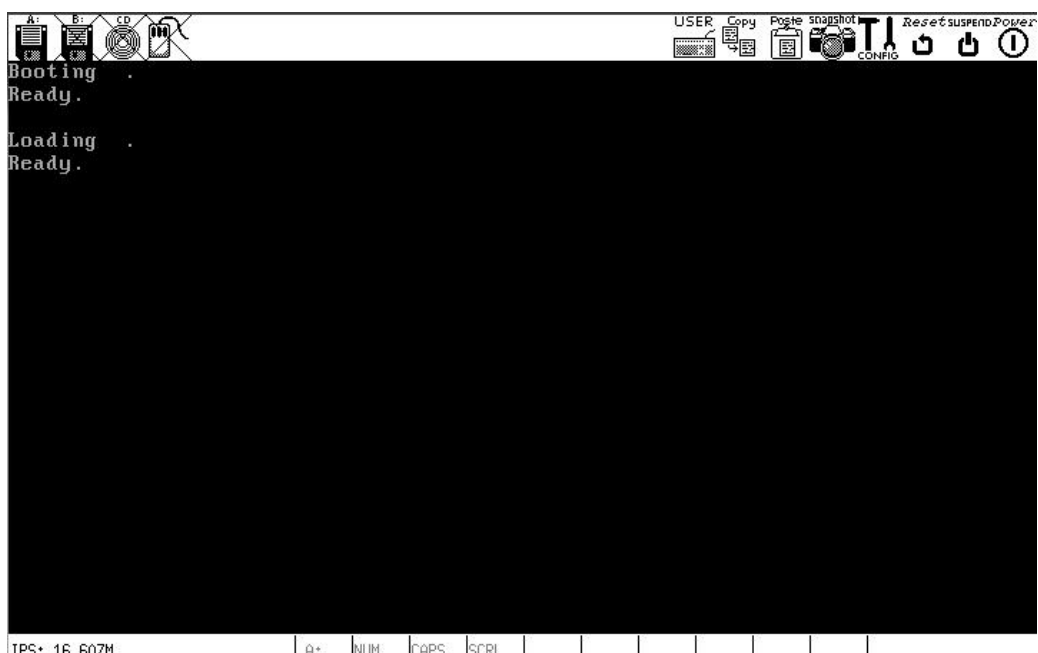


图 8.9 载入内核 Kernel

可以看到，程序成功地载入了内核 Kernel，而同时，笔者也在第五章学习了汇编语言和 C 语言的共同使用，即通过 ELF 格式，使 C 语言和汇编语言可以通过 PUBLIC、EXTERN、GLOBAL 等关键字互相调用，使未来的编程可以摆脱汇编语言，进入更直观的 C 语言。

```

A1 D1 L1
USB Lopy Mouse Keyboard Reset suspend Power
CONFIG

>booting .....
Ready.

Loading .....
Ready.

BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0009F000h 00000000h 00000001h
0009F000h 00000000h 00001000h 00000000h 00000002h
0000E000h 00000000h 00018000h 00000000h 00000002h
00100000h 00000000h 01EF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFC0000h 00000000h 00040000h 00000000h 00000002h

RAM size:01FF0000h
-----"cstart" begins-----
-----"cstart" finished-----
-----"kernel_main" begins-----
A0x0.^A0x1.^A0x2.^A0x3.^A0x4.^A0x5.^A0x6.^A0x7.^A0x8.^A0x9.^A0xA.^A0xB.^A0xC.^A0xD.^A0xE.^A0xF.^A0x10.^A0x11.^A0x12.^A0x13.^A0x14.^A0x15.^A0x16.^A0x17.^A0x18.^A0x19.^A0x1A.^A0x1B.^A0x1C.^A0x1D.^A0x1E.^A0x1F.^A0x20.^A0x21.^A0x22.^A0x23.^A0x24.^A0x25.^A0x26.^A0x27.^A0x28.^A0x29.^A0x2A.^A0x2B.^A0x2C.^A0x2D.^A0x2E.^A0x2F.^A0x30.^A0x31.^A0x32.^A0x33.^A0x34.^A0x35.^A0x36.^A0x37.^A0x38.^A0x39.^A0x3A.^A0x3B.^A0x3C.^A0x3D.^A0x3E.^A0x3F.^A0x40.^A0x41.^A0x42.^A0x43.^A0x44.^A0x45.^A0x46.^A0x47.^A0x48.^A0x49.^A0x4A.^A0x4B.^A0x4C.^A0x4D.^A0x4E.^A0x4F.^A0x50.^A0x51.^A0x52.^A0x53.^A0x54.^A0x55.^A0x56.^A0x57.^A0x58.^A0x59.^A0x5A.^A0x5B.^A0x5C.^A0x5D.^A0x5E.^A0x5F.^A0x60.^A0x61.^A0x62.^A0x63.^A0x64.^A0x65.^A0x66.^A0x67.^A0x68.^A0x69.^A0x6A.^A0x6B.^A0x6C.^A0x6D.^A0x6E.^A0x6F.^A0x70.^A0x71.^A0x72.^A0x73.^A0x74.^A0x75.^A0x76.^A0x77.^A0x78.^A0x79.^A0x7A.^A0x7B.^A0x7C.^A0x7D.^A0x7E.^A0x7F.^A0x80.^A0x81.^A0x82.^A0x83.^A0x84.^A0x85.^A0x86.^A0x87.^A0x88.^A0x89.^A0x8A.^A0x8B.^A0x8C.^A0x8D.^A0x8E.^A0x8F.^A0x90.^A0x91.^A0x92.^A0x93.^A0x94.^A0x95.^A0x96.^A0x97.^A0x98.^A0x99.^A0x9A.^A0x9B.^A0x9C.^A0x9D.^A0x9E.^A0x9F.^A0xA0.^A0xA1.^A0xA2.^A0xA3.^A0xA4.^A0xA5.^A0xA6.^A0xA7.^A0xA8.^A0xA9.^A0xAA.^A0xAB.^A0xAC.^A0xAD.^A0xAE.^A0xAF.^A0xB0.^A0xB1.^A0xB2.^A0xB3.^A0xB4.^A0xB5.^A0xB6.^A0xB7.^A0xB8.^A0xB9.^A0xBA.^A0xBB.^A0xBC.^A0xBD.^A0xBE.^A0xBF.^A0xC0.^A0xC1.^A0xC2.^A0xC3.^A0xC4.^A0xC5.^A0xC6.^A0xC7.^A0xC8.^A0xC9.^A0xCA.^A0xCB.^A0xCC.^A0xCD.^A0xCE.^A0xCF.^A0xD0.^A0xD1.^A0xD2.^A0xD3.^A0xD4.^A0xD5.^A0xD6.^A0xD7.^A0xD8.^A0xD9.^A0xDA.^A0xDB.^A0xDC.^A0xDD.^A0xDE.^A0xDF.^A0xE0.^A0xE1.^A0xE2.^A0xE3.^A0xE4.^A0xE5.^A0xE6.^A0xE7.^A0xE8.^A0xE9.^A0xEA.^A0xEB.^A0xEC.^A0xED.^A0xEE.^A0xEF.^A0xF0.^A0xF1.^A0xF2.^A0xF3.^A0xF4.^A0xF5.^A0xF6.^A0xF7.^A0xF8.^A0xF9.^A0xFA.^A0xFB.^A0xFC.^A0xFD.^A0xFE.^A0xFF

IPS: 9,469M A: NUM CAPS SCRL

```

笔者的实验仅仅完成了最简单的进程，而《ORANGE'S: 一个操作系统的实现》一书还在后面实现了更多的内容：多进程、系统调用、进程调度算法等等，图 8.11 为本章最后的内容（仅仅做了调试并阅读了代码）。

[illegible]

112

[illegible]

而在最后的第七章的内容中，笔者则完成了输入输出系统的实验，通过键盘中断处理程序与键盘控制器 8042 芯片和键盘编码器 8048 芯片进行交互，并且处理得到的 Make Code 及 Break Code 返回值，并且由特殊的函数解析扫描码，输出相应的字符，效果如图 8.13 所示。

```

A B C D
USER Copy Paste Snapshot Reset suspend Power
CONFIG

Booting .....
Ready.

Loading .....
Ready. _

BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0009F000h 00000000h 00000001h
0009F000h 00000000h 00001000h 00000000h 00000002h
000E8000h 00000000h 00018000h 00000000h 00000002h
00100000h 00000000h 01EF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFC0000h 00000000h 00040000h 00000000h 00000002h

RAM size:01FF0000h
----"cstart" begins-----
----"cstart" finished-----
----"kernel_main" begins-----
abc123  00040000h

```

113

而对于第七章后续的内容,笔者也进行了调试。在第七章后续的内容中,实现了多控制台,通过按下 Shift+F1/F2/F3 这三个按键可以进行三个控制台的切换,控制台 1 中进行时间刻系统调用的测试;控制台 2 中有 A、B 两个进程,进行多进程调试;控制台 3 中则进入输入输出测试,如图 8.14、图 8.15、图 8.16 所示。

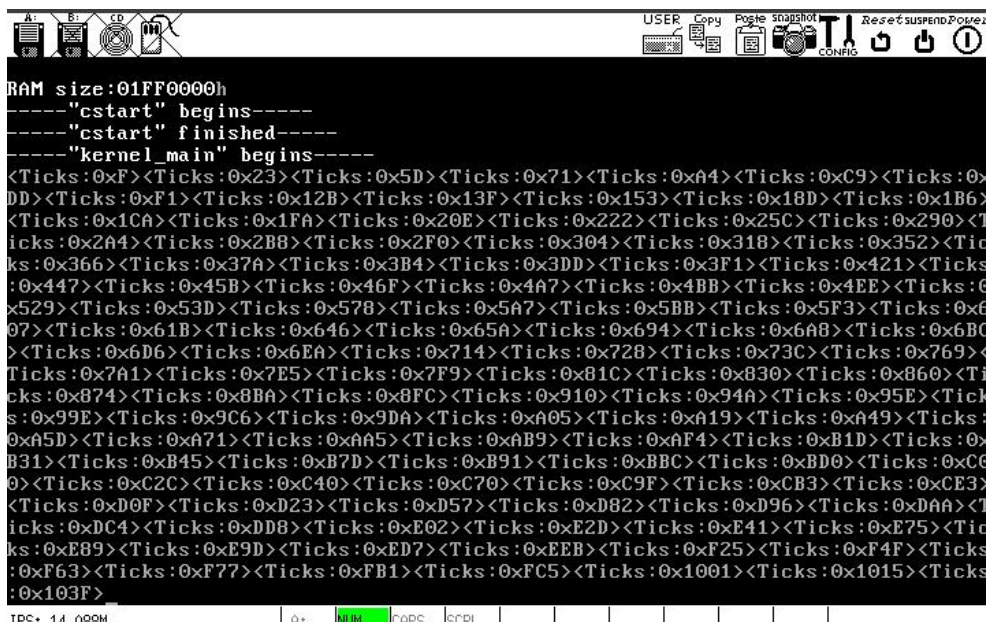


图 8.14 时间刻系统调用测试

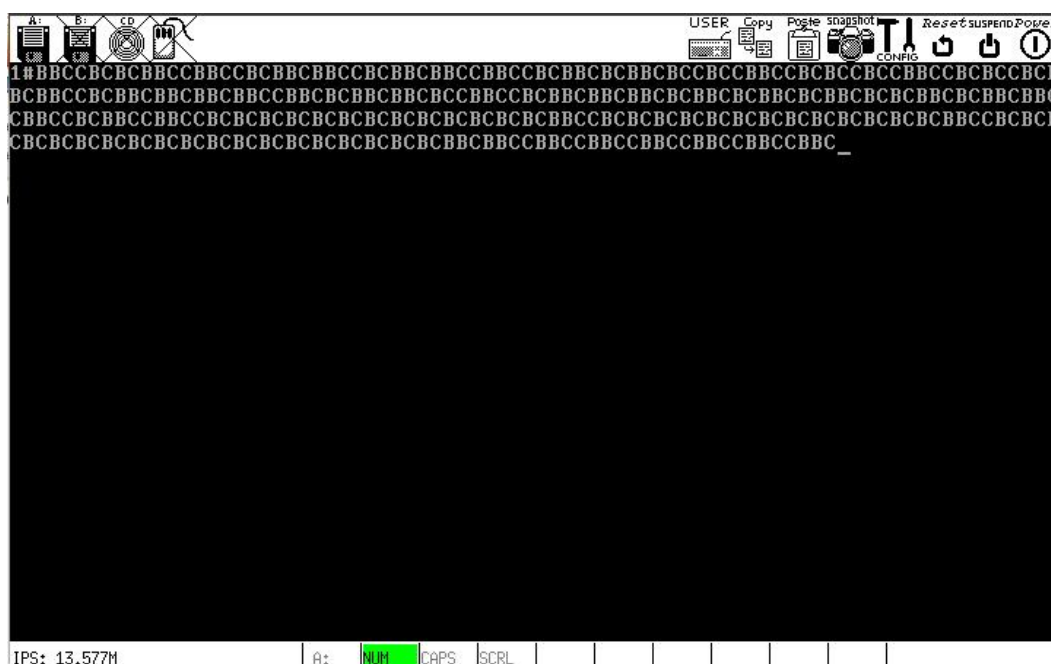


图 8.15 多进程测试



图 8.16 输入输出测试

同时，注意到本书配套代码对 TAB 键输入没有响应，笔者通过在两个 switch 语句中增添对 TAB 键的响应，使其输出两个空格。功能改进的效果如图 8.17 所示。

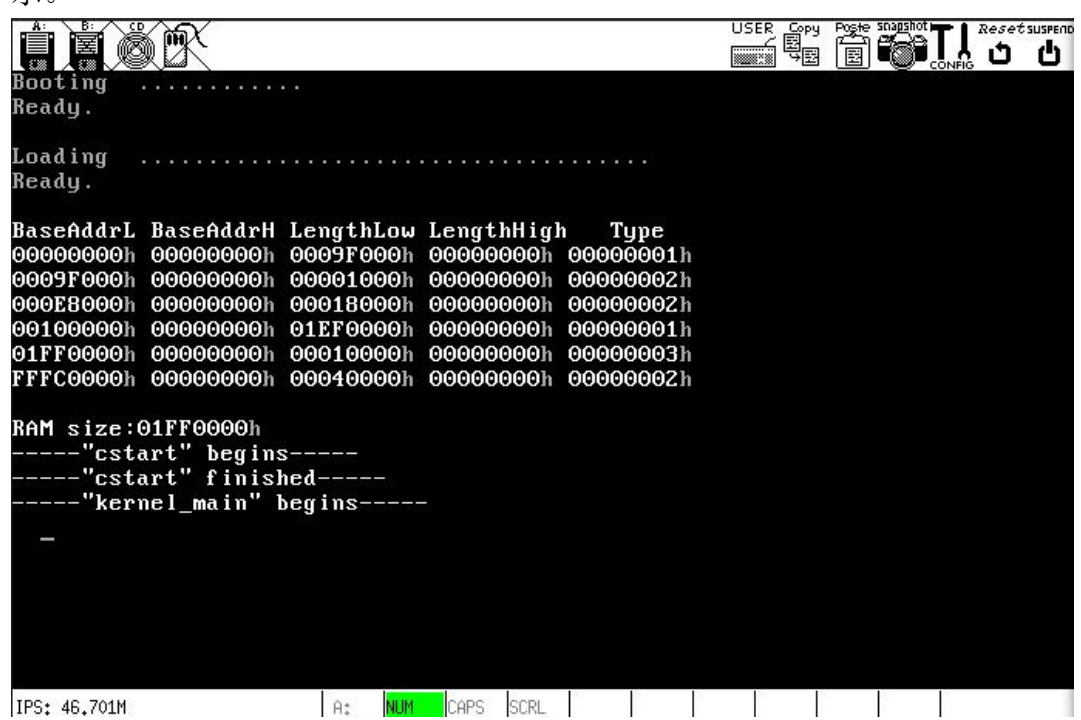


图 8.17 改进后的调试结果

8.2.2 实验心得

在本次操作系统复现实验中，笔者选择《ORANGE'S: 一个操作系统的实现》一书，实现了其中的源代码，并对其他相关的实验进行了调试。最后，就实验结果而言，实现了一个具有引导 boot 和 Loader 以及内核 Kernel 的操作系统，并且完成实模式到保护模式的转换以及实现中断和输入输出以及多进程功能。

通过该实验，笔者主要有以下收获：

第一，这个实验使笔者对 CPU 的硬件有了初步的了解，为了实现操作系统，就难免要与硬件进行交互，而在本实验中，则主要与 CPU 的中断控制芯片 8259A、时钟控制芯片 8253、键盘控制芯片 8042 等芯片进行了控制，使笔者对这些芯片有了初步的了解；

第二，这个实验加深了笔者对操作系统的了解，在本实验中，实现了操作系统的引导磁盘 Boot、操作系统内核 Kernel、多进程、输入输出、多控制台、进入保护模式、分页机制等方面的内容，使笔者对操作系统的这些内容有了更加深刻以及直观的了解，不仅仅知道它们的作用，也深入地了解了它们的内部实现；

第三，这个实验使笔者对 x86 的汇编语言有了初步的了解，在本实验设计的前半段，使用了大量的汇编语言，非常的繁复，虽然至今笔者对许多代码还有着困惑，但通过阅读这些代码，还是对汇编语言有了一个初步的了解，学习了许多汇编代码的意义和功能；

第四，这个实验使笔者对虚拟机和 Linux 操作系统有了一个初步的认识，在过去，仅仅只使用过 Windows 操作系统，也没有使用过虚拟机软件，而这次的实验无疑给了一个很好的机会，去尝试使用虚拟机并且尝试使用 Linux 操作系统的 Ubuntu 操作系统，尝试了 Linux 操作系统并且了解了一些基本的常用指令。

第五，这个实验增强了笔者的动手能力，培养了耐心，虽然这次实验仅仅实现了全部代码中的一部分，但这也已经是非常大的工作量，并且出现了大量的错误，在敲代码和 debug 的过程中，无疑是大大地提高了实际动手能力以及增强了耐心。

第六，这个实验拓宽了笔者对计算机有关知识的认识，在经历完这个实验后，笔者不仅仅是对操作系统，也对其他有关计算机的知识有了一些了解，极大地拓宽了知识面。

第七，这个实验增强了笔者检索信息找到答案的能力，在这个实验中，出现了海量的问题，而这些问题在书本上根本找不到答案，除了询问老师、同学以及 ChatGPT 外，也查阅了大量百度百科、CSDN、博客园、GitHub 上的有关资料，以便能解决实验中的问题，使得信息检索能力有所提升。

虽然这个实验的难度较大，但在完成这个实验之后笔者对操作系统的了解、对计算机相关知识的了解以及实际动手能力都有所提升。虽然一开始笔者有很大的畏难情绪，但经过一个月的反复琢磨和折腾，最终还是大致完成了这个实验，也相信笔者在这个实验中学习到的知识和方法将对未来的保研面试、学习和工作有利。

此次实验中，笔者了解了操作系统的各要素及涉及开发操作系统的各方面，从一个最简单的引导扇区开始，逐渐完善代码，扩充了保护模式、内核、进程、输入/输出系统等功能，实现了一个小的操作系统。虽然过程中有很多的艰难险阻，但还是一路走来解决了这些困难。这个过程中，非常感谢曾老师对笔者的指导和帮助。

9 参考文献

- [1]于渊. 《ORANGE'S: 一个操作系统的实现》[M] 第二版 北京: 电子工业出版社, 2010 年: P1-P278
- [2]郑鹏 曾平 金晶. 《计算机操作系统》[M] 第二版 武汉: 武汉大学出版社, 2014 年: P1-P206

10 教师评语评分

评语： _____

评分： _____

评阅人：

年 月 日

（备注：对该实验报告给予优点和不足的评价，并给出百分制评分。）