



浙江大学

基于 Bochs 的分时多任务调度器

陈思宇，陈璟洲，THACHAN SOPHANYOULY(尤利)

2017-08-20

摘 要

调度器可以看作是一个简单的操作系统，允许以周期性或单次方式来调用任务。从底层的角度看，调度器可以看作是一个由许多不同任务共享的定时器中断服务程序，因此，只需要初始化一个定时器，而且改变定时的时候通常只需要改变一个函数。此外，无论需要运行 1 个，10 个还是多个不同的任务，通常都可以使用同一个调度器完成。随着计算机的发展，我们比较容易能够使用高级程序语言来实现调度器。那么使用一个低级程序语言像汇编语言能不能实现一个调度器。因此本次实验我们将要基于 Bochs，一个模拟器，在 8086 的实模式以及 80386 的保护模式下利用汇编语言实现两个简单任务调度器。

关键词：Bochs，调度器，实模式，保护模式

目 录

第一章 绪论	4
1.1 虚拟计算机 Bochs	4
1.2 Bochs 的安装	4
1.3 Bochs 的使用	5
第二章 实验主要内容	7
2.1 研究思路	7
2.2 研究过程与实现详解	8
2.3 可重入程序调度器	9
2.4 2 级位置无关的内核加载器	14
2.5 2 级加载的 C 语言内核	17
第三章 保护模式	26
3.1 描述符 Descriptor	26
3.2 选择子 Selector	28
3.3 80386 寄存器结构	29
3.4 保护模式下的寻址	30
3.5 特权权	31
3.6 2 级加载的保护模式下的调度器	33
第四章 总结	37
参考文献	38

第一章 绪论

1.1 虚拟计算机 Bochs

即便没有听说过虚拟计算机，你至少应该听说过磁盘映像。如果经历过 DOS 时代，你可以就曾经用 HD-COPY 把一张软盘做成一个 .IMG 文件，或者把一个 .IMG 文件恢复成一张软盘。简单来讲，它相当于运行在计算机的小计算机。在介绍 Bochs 及其他工具之前，需要说明一点，这些工具并不是不可或缺的，介绍它们仅仅是为了提供一些可供选择的方法，用以搭建自己的工作环境。但是，这并不代表这一章就不重要，因为得心应手的工具不但可以愉悦身心，并且可以起到让工作事半功倍的功效。下面就从 Bochs 开始介绍。

我们先来看看 Bochs 是什么样子的，请看图 1.1 这一个屏幕截图。窗口的标题栏一行 “Bochs x86 emulator, <http://bochs.sourceforge.net/>” 明白无误地告诉我们，这仅仅是个 “emulator” —— 模拟器而已。在本文中我们把这种模拟器成为虚拟机，因为这个词使用得更广泛一些。不管是模拟还是虚拟，我们要的就是它，因为有了它我们不再需要频繁地重启计算机，即便程序有严重的问题，也丝毫伤害不到你的计算机。更加方便的是，可以用这个虚拟机来进行操作系统的调式。

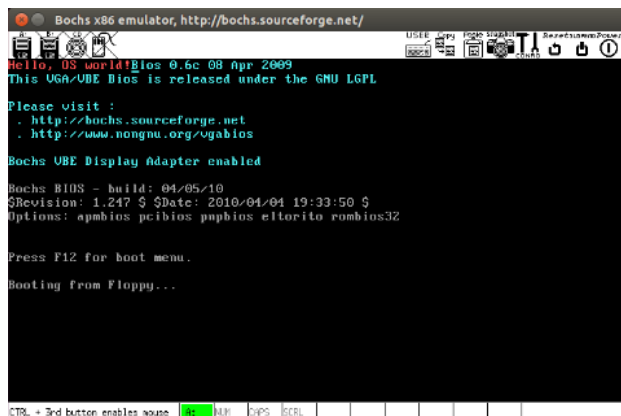


图 1.1, Linux 中的 Bochs

1.2 Bochs 的安装

就像大部分软件一样，在不同的操作系统里面安装 Bochs 的过程是不同的，在 Window 中，最方便的方法就是从 Bochs 的官方网站获取安装程序来安装（安装时不妨将 “DLX Linux Demo” 选中，这样你可以参考它的配置文件）。在 Linux 中，不同的发行版（distribution）处理方法可能不同。比如，如果你用的是 Debian GNU/Linux 或其近亲（Ubuntu），可以使用这样的命令：

```
sudo apt-get install vgabios bochs bochs-x bximage
```

敲入这样一行命令，不一会儿就装好了。

很多 Linux 发行版都有自己的包管理机制，不如上面这行命令就使用了 Debian 的包管理命令，不过这样安装虽然省事，但有个缺点不得不说，就是默认安装的 Bochs 很可能是没有调式功能的，这显然不能满足我们的需要，所以最好的方法还是从源代码安装，源代码同样位于 Bochs 的官方网站，假设你下载的版本是 2.3.5，那么安装过程差不多如下：

```
tar vxzf bochs-2.3.5.tar.gz
cd bochs-2.3.5
./configure --enable-debugger --enable-disasm
make
sudo make install
```

注意“./configure”之后的参数便是打开调式功能的开关。在安装过程中，如果遇到任何困难，不要惊慌，其官方网站有详细的安装说明。

1.3 Bochs 的使用

上面有提到软盘，那么软盘究竟是什么？既然计算机都可以“虚拟”，软盘当然也可以。在刚刚装好的 Bochs 组件中，就有一个工具叫做 bximage，它不但可以生成虚拟软盘，还能生成虚拟硬盘，我们也称它们为磁盘映像。创建一个软盘映像的过程如图 1.2 所示：

```
=====
                        bximage
          Disk Image Creation Tool for Bochs
      $Id: bximage.c,v 1.34 2009/04/14 09:45:22 sshwarts Exp $
=====

Do you want to create a floppy disk image or a hard disk image?
Please type hd or fd. [hd] fd

Choose the size of floppy disk image to create, in megabytes.
Please type 0.16, 0.18, 0.32, 0.36, 0.72, 1.2, 1.44, 1.68, 1.72, or 2.88.
[1.44]
I will create a floppy image with
  cyl=80
  heads=2
  sectors per track=18
  total sectors=2880
  total bytes=1474560

What should I name the image?
[a.img]

Writing: [] Done.

I wrote 1474560 bytes to a.img.

The following line should appear in your bochsrc:
  floppy0: image="a.img", status=inserted
```

图 1.2, bximage, 创建一个软盘映像

在上面只有一个地方没有使用默认值，就是被问到创建硬盘还是软盘映像的时候，就输入了“fd”。

完成这一步骤之后，当前目录下就多了一个 a.img，这便是软盘映像了。所谓映像者可以理解为原始设备的逐字节复制，也就是说，软盘的第 M 个字节对应映像文件的第 M 个字节。

现在我们已经有了“计算机”，也有了“软盘”，是时候将引导扇区写进软盘了。可以使用 dd 命令：

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

注意这里多用了参数“conv=notrunc”，如果不用它的话软盘映像文件 a.img 会被截断（truncated），因为 boot.bin 比 a.img 要小。

现在一切准备就绪，只差一个 Bochs 的配置文件。为什么要有配置文件呢？因为我们需要告诉 Bochs，希望我们的虚拟机是什么样子的。比如，内存多大、硬盘映像和软盘映像都是哪些文件等内容。图 1.3 就是一个 Linux 下的典型配置文件例子。

```
#Configuration file for Bochs

#how much memory the emulated machine will have
megs: 32

#filename of ROM images
romimage: file=/usr/local/share/bochs/BIOS-bochs-latest
vgaromimage: file=/usr/local/share/bochs/VGABIOS-lgpl-latest

#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/local/share/bochs/keymaps/x11-pc-us.map
```

图 1.3, bochsrc 示例

可以看到，这个配置文件本来就不长，除去注释之后内容更少了，而且很容易理解，字面上稍微不容易理解的只有 romimage 和 vgaromimage，它们指定的文件对应的其实就是机器的 BIOS 和 VGA BIOS，操作时需要确保它们的路径是正确的，不然过一会儿虚拟机启动时可能会被提示“couldn't open ROM image file”。除了之外还要注意 floppya 一项，它指定我们使用哪个文件作为软盘映像。现在一切准备就绪，是时候启动了，输入命令：

```
bochs -f bochsrc
```

一个回车，结果就如图 1.1 所示。

第二章 实验主要内容

不借助任何外部代码，不借助《30 天教你从头写 XXX》类书籍，不借助现有 OS 的工程思路，不借助网络上博客的“讲解”，进行如下工作：

1. 阅读英特尔 CPU 产品说明书，掌握 CPU 工作方式、指令的详尽功能；
2. 阅读 BIOS 产品说明书，全面掌握 BIOS 的功能，掌握使用要点与调用方法；
3. 阅读 GAS 软件产品说明书，全面掌握 GAS 的汇编语法与工程使用；
4. 阅读 NASM 软件产品说明书，全面掌握 NASM 的汇编语法与工程使用；
5. 阅读各平台 ABI 说明书，选择性地大致掌握主流二进制文件的结构；
6. 编写能够实现输出的 HELLOWORLD 程序。
7. 编写能对简单可重入客户程序进行调度的调度器。
8. 编写 2 级加载的内核加载器。
9. 编写能实现位置无关运行的 2 级内核加载器。
10. 编写 2 级加载的 C 语言内核，并实现调度、维护，提供简单的系统调用与服务。
11. 编写 2 级加载的保护模式下的调度器。

如有雷同，绝非巧合。因为工程上的雷同大都因为框架的限制与规定。

2.1 研究思路

容易想出来的模式有两种：

- 1，FIFO 模式（管道调度）
- 2，仲裁模式（事件调度）

我们采用明显更为适合的仲裁模式。

CPU 上电基本环境为 0x7c00 开始运行，cs=0x0000 ip=0x7c00。内存结构如下图所示，共计 1MB。A20 地址线未打开，如果访存超 1MB，则会被环回 0 地址。

Address Range	Function
000000H-09FFFFH	640 KB conventional RAM
0A0000H-0BFFFFH	VGA DRAM (typically on the PCI backplane)
0C0000H-0C7FFFFH	VGA ROM (typically on the PCI backplane)
0C8000H-0DFFFFH	Expansion ROM
0E0000H-0EFFFFH	System BIOS extensions
0F0000H-0FFFFFFH	Phoenix system BIOS

2.2 研究过程与实现详解

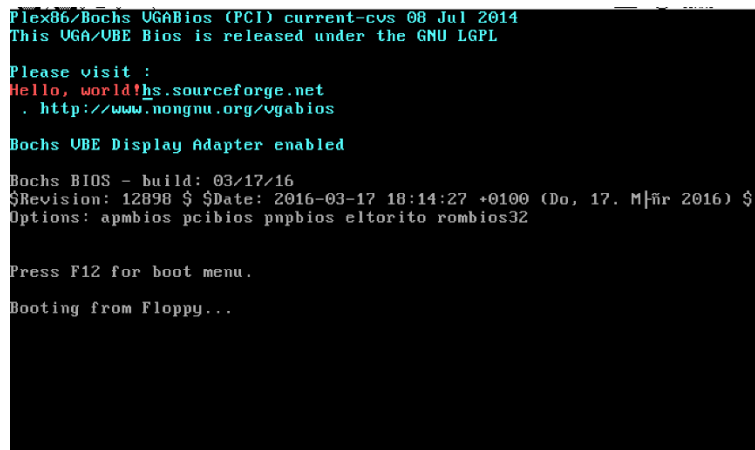
实现打印与光标控制功能，直接覆盖到 bios 信息上面。首先展示我们研究过程中使用的编译方法，如下图所示。

```
chens-MacBook:asm chen$ make all target=learning/hello
nasm learning/hello.s -o learning/hello.o
dd if=learning/hello.o of=learning/hello.img bs=512 count=
1 conv=notrunc
1+0 records in
1+0 records out
512 bytes transferred in 0.000022 secs (23091222 bytes/sec
)
chens-MacBook:asm chen$
```

因为该项目对编译的需求比较复杂，总结起来包括：

1. 调用不同的汇编编译器处理 S 源代码
2. 处理并抽象汇编器的一大堆参数，保存多种汇编方案
3. 读写镜像文件
4. 打包与保存，简单的版本管理
5. 处理 C 语言的编译生成、编译器参数
6. 处理链接器的参数
7. 处理 ABI 格式并进行重构拼接
8. 保存、显示功能提示笔记和帮助

第一个项目实现时，Makefile 功能还很简单，只用了一小部分功能。在此实验之后的研究中，Makefile 的功能逐渐被填充增加，以满足研发过程中出现过的乱七八糟的需求。



```
Flex86/Bochs VGABios (PCI) current-cvs 08 Jul 2014
This VGA/VBE Bios is released under the GNU LGPL

Please visit :
Hello, world!hs.sourceforge.net
. http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

Bochs BIOS - build: 03/17/16
$Revision: 12898 $ $Date: 2016-03-17 18:14:27 +0100 (Do, 17. Mär 2016) $
Options: apmbios pcibios pnpbios eltorito rombios32

Press F12 for boot menu.
Booting from Floppy...
```

图 2.1, Hello, World 的运行结果（红色）

主要代码：


```

1  org 07c00h
2  mov ax, cs
3  mov ds, ax
4  mov es, ax
5  call DispStr
6  jmp $
7  DispStr:
8  mov ah, 02h
9  mov bh, 0
10 mov dh, 4h
11 mov dl, 4h
12 int 10h
13 lea ax, [STRH]
14 mov bp, ax
15 mov cx, 13
16 mov ax, 01301h
17 mov bx, 000ch
18 mov dl, 0
19 int 10h
20 ret
21 STRH db "Hello, world!"
22 times 510-($-$$) db 0
23 dw 0xaa55
24

```

首先将地址翻译的预处理指针定位到 7c00 位置，设置代码段与数据段相同，便于索引数据。

VGABIOS 将功能共享内存映射到显存上，主板 BIOS 再将显存映射到主存上，并根据 VGA 型号在内存中写入控制程序。这里利用的是 10h 号中断的 2h 偏移功能包装了一个 DisplayString 函数。

2.3 可重入程序调度器

这次编写的调度器尝试调度两个简单的打印程序，一个打印 A，另一个打印 B，两者交替运行。程序运行效果如图 2.2 所示。

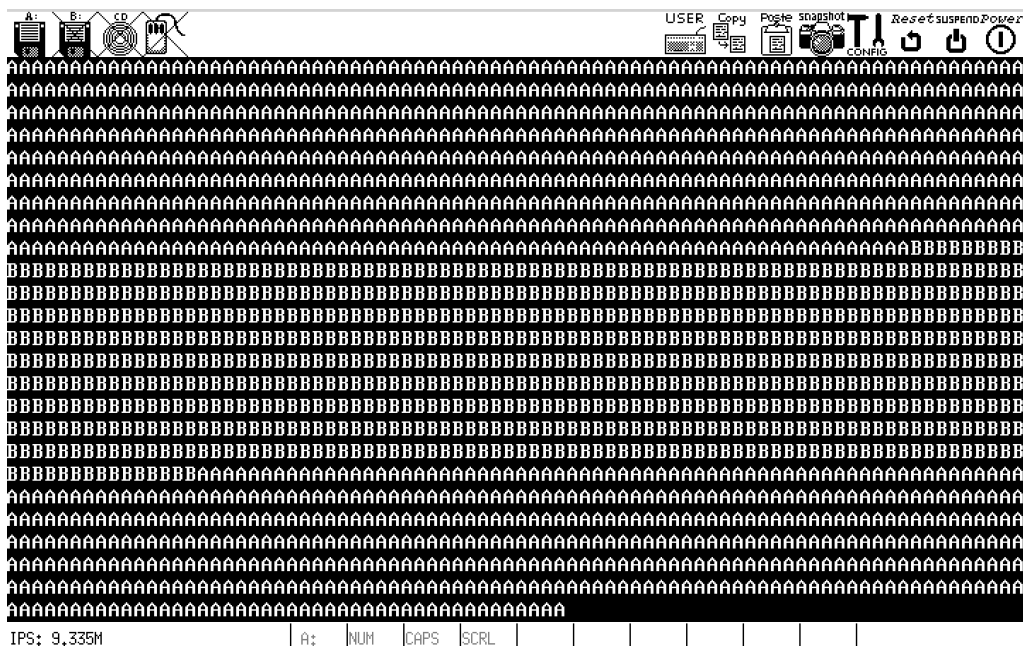


图 2.2，调度两个简单的打印程序

程序代码说明：

```
1  org 07c00h
2  mov ax, cs
3  mov ds, ax
4  mov es, ax
5  ;; set interrupt
6  push ax
7  mov al, 0x04
8  mov ah, 0x1c
9  mul ah ;; ax=1ch*4
10 lea dx, [foo]
11 mov di, ax ;; di=1ch*4
12 pop ax ;; ax=CS
13 mov [di], dx ;; put foo IP
14 add di, 2
15 mov [di], ax ;; put foo CS
16 ;; INT
17 sti
18 cld
19 call ClearS
20 fnop
21 jmp fnop
```

基于上一份代码主要的更改就是“中断劫持”。我们的思路是通过时钟中断唤醒调度程序。修改中断服务程序的返回地址，使得中断结束后，返回到一个我们指定的位置。

我们要劫持的中断号是 1c，该中断是由 08h 中断触发的软中断，08h 是由 8259 级联片触发的硬件中断，8259 从晶体振荡钟获得时钟触发的电气脉冲。

鉴于每个中断向量占 4 字节，其中低 2 子节为 IP，高 2 子节为 CS，我们将 1ch 乘 4 获得要修改的内存首地址，将该处内容改写为我们的调度器地址即可完成劫持。

图 2.3 展示的是两个被调度的客户程序。

```
23 main1:
24 main1lp
25 mov al, 41h ; 'A'
26 call Sprintal
27 jmp main1lp
28 ret
29
30 main2:
31 main2lp
32 mov al, 42h ; 'B'
33 call Sprintal
34 jmp main2lp
35 ret
```

图 2.3，两个被调度的客户程序

之后，我们实现了调度器的逻辑，最初有两种考虑，分别为 2 状态调度与 4 状态调度，两种状态调度模式如下：

1. 状态 1: 应该运行 main1
2. 状态 2: 应该运行 main2

3. 永远在 1, 2 之间切换跳转。

四种状态调度模式如下:

1. 状态 1:应该启动 main1
2. 状态 2:main1 被中断, 启动 main2
3. 状态 3:main2 被中断, 还原 main1
4. 状态 4:main1 被中断, 还原 main2
5. 稳定在 3, 4 之间切换跳转。

我们最终选择了四种状态的调度逻辑, 代码如下。

```
38 sched:
39 push ebp
40 mov ebp,esp
41 push eax
42 push edx
43 push ecx
44 mov ax,[SCHED_SAVE]
45
46 cmp ax,0
47 jnz .state0end
48 .state0 ;;no task is loaded
49 lea eax,[main1]
50 mov word [dword ebp+16],ax
51 mov ax,1
52 jmp .savestate
53 .state0end
54
55 cmp ax,1
56 jnz .state1end
57 .state1 ;;task 1 is interrupted, task2 not loded
58 lea esi,[dword ebp+4]
59 lea edi,[TASK1]
60 mov ecx,18
61 rep movsb
62 lea eax,[main2]
63 mov word [dword ebp+16],ax
64 mov ax,2
65 jmp .savestate
66 .state1end
67
68 cmp ax,2
69 jnz .state2end
70 .state2 ;;task 1 is saved, task 2 is interrupted
71 lea esi,[dword ebp+4]
72 lea edi,[TASK2]
73 mov ecx,18
74 rep movsb
75 lea esi,[TASK1]
76 lea edi,[dword ebp+4]
77 mov ecx,18
78 rep movsb
79 mov ax,3
80 jmp .savestate
81 .state2end
82
83 cmp ax,3
84 jnz .state3end
85 .state3 ;;task 2 is saved, task 1 is interrupted
86 lea esi,[dword ebp+4]
87 lea edi,[TASK1]
88 mov ecx,18
89 rep movsb
90 lea esi,[TASK2]
91 lea edi,[dword ebp+4]
92 mov ecx,18
93 rep movsb
94 mov ax,2
95 .state3end
96
97 .savestate
98 mov [SCHED_SAVE],ax
99 pop ecx
100 pop edx
101 pop eax
102 pop ebp
103 iret
```

当中断触发，进入调度器后，首先保护父过程的栈底地址 `ebp`，将上一次的状态从内存读取到 `ax` 中。之后将 `ax` 与 0-3 比较，实现 C 语言中 `switch` 的功能。由于 INTEL 芯片中断的行为如下：

1. 查看 `int` 偏移值是否合法
2. 检查栈够 6byte
3. `push(eflags[0:15]); 2bytes`
4. 关中断，清 `trap`，清 `AC`
5. `push(CS) 2bytes`
6. `push(IP) 2bytes`

故在状态的处理中，会将上次程序的 `flag`、`IP` 从栈底后 16byte 处读取出来，因为最初 `main` 被中断时，其地址保存在 `0x08` 中断的 `ebp+0` 到 `ebp+5` 的位置上，换算到第二次调用，就是 $6*2+size(ebp)=16$ 。根据此地址，可以从此处取出上一次的状态，并写入想要的状态。

比较完成后，将新的状态号写入 `ax`，跳转到结束保存位置 “`savestate`”，该位置的代码会将 `ax` 写回内存，退出中断，返回到指定位置。

同时，我们在这一阶段也实现了对 CRT 硬件的基本控制。首先是屏幕清除函数，代码如下。

```
105 ClearS:      ; clear screen
106 mov al,0h
107 ScrollS:     ; al=lines up, when al=0, clear!
108 ;; destory AX! others safe
109 push ebp     ; clear will destory ebp
110 push esp
111 push ax
112 push cx
113 push dx
114 mov ah,06h
115 mov cx,0h
116 mov dh,[CRT_ROW]
117 mov dl,80
118 int 10h
119 pop dx
120 pop cx
121 pop ax
122 pop esp
123 pop ebp
124 ret
```

它是用屏幕翻滚中断来实现的，当翻滚为 0 时，则清屏，光标位置不变。这里我们发现用双头方式定义默认参数构造函数非常方便，算是汇编编程领悟到的小 `trick`。这个 `Scroll` 功能行为其实比较不完善，因为会破坏 `eax`

与 ebp、ebx 寄存器，所以必须要预先保存。之后我们从内存位置取出行列值，发送命令让 int10-06 写入显存。

每次写入显存后，我们都要更新光标的位置，让它指向下一个字符块，如果到达行尾，则光标应该向下一行，然后回车；如果整个屏幕都已写满，则应该向上滚动屏幕，然后写在 CRT 最后一行，这样才能实现平时我们所熟悉的字符界面。函数 cursor_deal 如下。

```
167 cursor_deal:
168 ;; deal with dh,dl
169 ;; safe to other regs
170 push ax
171 cmp dl,80
172 jb CD_COLS_GOOD
173 ;; if col is too large
174 mov dl,0 ;reset col
175 inc dh ;dh++
176 jmp CD_ROWS_DEAL
177 CD_COLS_GOOD
178 inc dl ;dl++
179 CD_ROWS_DEAL
180 cmp dh,[CRT_ROW]
181 jbe CD_ROWS_GOOD
182 mov al,1
183 call ScrollS
184 mov dh,[CRT_ROW] ;;if row>ROW, fix row=ROW
185 CD_ROWS_GOOD
186 pop ax
187 ret
```

我们的 CRT 是标准设备，大小为 80x25。初始光标在 (0,0) 位置，每次写完后，我们都要计算光标下一个可写的坐标，并保存该坐标到内存。必要时，会滚动屏幕。

下面的函数实现了打印单个字符到显示器的功能，通过修改参数，也可以实现打印多彩色字符。

```
146 DispChar: ;;display A by default
147 mov al,41h ;; 'A'
148 DispAL: ;; put ascii hex into al to print char
149 push ebp
150 push esp
151 push ax
152 push bx
153 push cx
154 mov ah,09h
155 mov bh,0
156 mov bl,0fh
157 mov cx,1
158 int 10h
159 pop cx
160 pop bx
161 pop ax
162 pop esp
163 pop ebp
164 ret
```

这里我们也使用了双头函数来包装默认功能与详细功能的不同入口。之后我们编写了光标设置函数。该函数接收行列参数来放置光标。代码如下所示。

```
126 ResetCursor:
127 mov dx,0
128 SetCursor:
129 ; set cursor pos
130 ; row=dh col=dl
131 ; safe except for dx
132 push ebp
133 push esp
134 push ax
135 push bx
136 mov ah, 02h
137 mov bh, 0 ;; page no
138 int 10h
139 pop bx
140 pop ax
141 pop esp
142 pop ebp
143 ret
```

在实验的过程中，我们发现这个简单的调度器当陷入到我们自己编写的“系统调用”函数中时，如果被中断切出，则会因为已经使用的栈空间而造成内存泄漏。所以，我们在此确定了一个非抢断的实现方案，使得陷入 API 后进程不得被抢断。打印的服务函数如下。

```
189 Sprintal:
190 ;; put ascii code in AL to print
191 ;; safe to other regs
192 cli
193 push dx
194 mov dx,[DX_SAVE]
195 call SetCursor
196 call DispAL
197 call cursor_deal
198 mov [DX_SAVE],dx
199 pop dx
200 sti
201 ret
```

该函数在进入时关中断，离开时开中断，避免被调度而泄漏内存。

2.4 2级位置无关的内核加载器

因为 BIOS 的规定与限制，首次读盘只能加载 0 柱面 0 磁头 1 扇区 512byte 的内容，这对于一个稍微复杂一点的启动程序来说已是远远不够的，所以实现从其他存储介质中读取代码复制到内存中运行是不可避免的。因此，我们计划将 bootloader 写入软驱的第一个扇区，再编写一个小程序放到硬盘中等待被 bootloader 加载。

我们曾想尝试直接从硬盘中读取第一扇区作为 bootloader，读取之后的十个扇区作为 mini-kernel，但是我们经历数次挫折后发现，由于 bochs 已知的 BUG，从 HD 中读首扇区后再次读取时常会发生错误，而软盘则不会；另外考虑到我们应该实践练习一下对多种存储设备进行读写，所以最终选择了这种复杂的组合。

这次编写的代码中，bootloader 的主干逻辑如下所示。

```
19  call ClearS
20  call hdread
21  jmp dword B00TS ;0x07e0:0x0000
```

首先清空屏幕，读取 mini-kernel 到 0x7e00 位置，然后跳转到新加载的代码处运行。

一般来说，编译代码时必须制定一个代码起始的默认地址，这个数值是必须指定的，用来给所有的 label、call、jmp 或内存访问编址。对于 bootsection，我们根据 BIOS 说明书已知其会被复制到 0x7c00 位置，所以我们可以显式用 org 指定。但是对于客户程序或者可能被升级的自写代码来说，编写者并不知道自己的代码可能被 bootloader / kernel 加载到什么位置上去，所以操作系统必须支持把任意编写的代码正确运行的功能，也就是 PIC/PIE(Position Independent Code/ Position Independent Executable)。

我们根据阅读 INTEL 说明书地址翻译部分学习的知识，认为段地址+偏移值的机制很好的支持了这种特性，因为地址的翻译是基于 eip 计算的逻辑地址，也称偏移值。我们的设想是将代码加载到 16B 对齐的位置，改变 CS 值，使得地址翻译正确进行。这样，客户的程序只需要从 0 编址就可以了。

这里的机制比较复杂，有一些误区，比如 PIE 机制可能被人误解为程序中不出现绝对的内存寻址，比如 gcc 的编译选项 fpie。实际上，任何程序都不能避免使用绝对值来表示地址，例如 C 语言的函数指针，必须是关于 CS 的偏移值。如果 CS 不改变的话，翻译一定会出错。所以即使在 gcc 中选了 fpie，也无法保证这代码就是理论上地址无关的。地址无关代码的支持是编译器与内核加载器紧密联系才能实现的特性。我们没有阅读 gcc 或 linux 的工程代码，不过我们猜想他们用的 PIE 机制应该和我们探索的极为相似。

接下来我们将主体框架拆分讲解，首先是读取硬盘的代码。代码如下所示。

```

25  hread:
26  push ebp
27  push esp
28  push ebx
29  mov ah,0
30  mov dl,0
31  int 13h
32  ;cmp ah,0
33  ;jnz hread ;; force reset
34  mov ax,0x7e0
35  mov es,ax
36  mov bx,0
37
38  mov ah,02h
39  mov al,1 ;blocks
40  mov ch,0 ;cylinder
41  mov cl,1 ;sector
42  mov dh,0 ;head
43  mov dl,0 ;ata0
44  ;les bx,[B00TS]
45  int 13h
46  pop ebx
47  pop esp
48  pop ebp
49  ; jc hread
50  ret

```

之后，我们把打印相关的代码统统移到 mini-kernel 里，这使得 bootloader 的大小减为原先的二分之一。Mini-kernel 的主干代码如下所示。

```

1  cli
2  cli
3  mov ax,cs
4  mov ds,ax
5  mov es,ax
6
7  call ClearS
8  call ResetCursor
9  mov al,43h ;'C'
10 call DispAL
11 jmp $

```

该部分代码是依照段首 0x00 为准编译的，具有通用性，因为编译器生成的 obj 都是从 0 编址。代码在逻辑上首先关中断，然后不管之前的 cs / ds 如何。统一将代码段数据段扩展数据段都设置为代码段。然后调用 mini-kernel 里的清屏函数，如果位置无关工作正常的话，屏幕上的 BIOS 信息将被清除。紧接着调用光标设置与字符打印函数打印一个 ‘C’ 字母，如果工作正常，就会在屏幕第一个字符块位置打印出该字符。该程序结果如图 2.4。

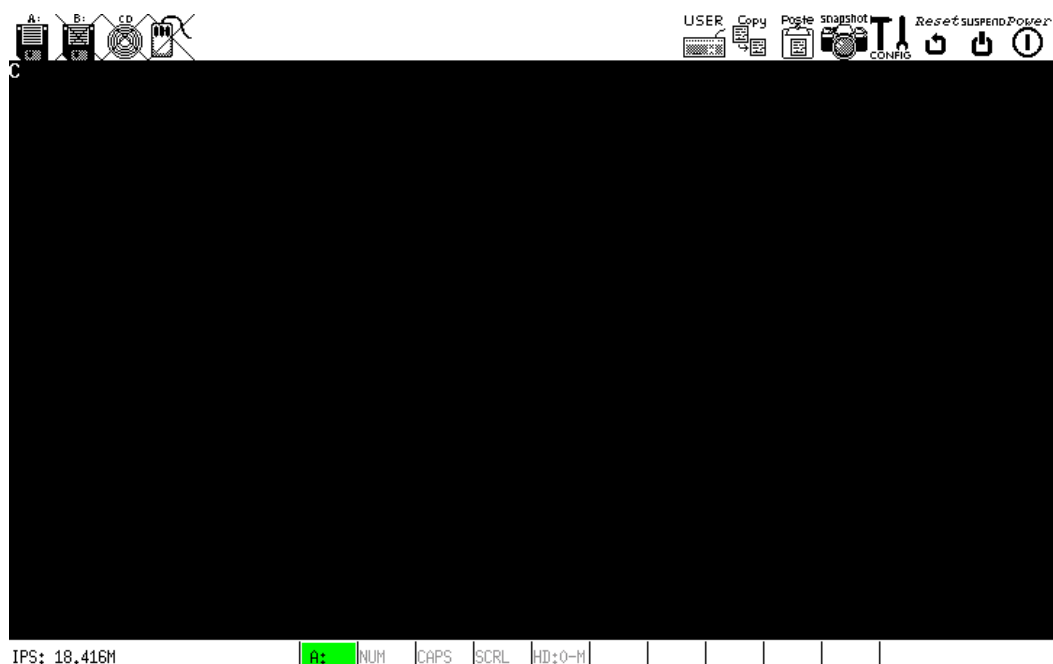


图 2.4，调用光标设置与字符打印函数打印一个‘C’字母

可见我们设计的 PIE 机制工作正确。

2.5 2 级加载的 C 语言内核

在这里我们的设计是让 bootloader 尽量小，装载完毕后直接跳转。而把中断处理、进程调度、系统服务等代码全都放进 mini-kernel 里面用 C 实现。这里的关键点有三个，分别是符号位置、代码格式、opcode 格式。

符号位置指的是我们如何定位终端服务函数，这样才能在 bootloader 中劫持中断。因为在之后的工作中我们会需要将内核改为保护模式下执行，所以必须在 bootloader 中设置中断。这时我们就需要详细知晓交叉编译常常遇到的 ABI 格式转换问题也就是代码格式，同时，原 ABI 中有很多没有用的结构，比如图 2.5 所示的 ELF 格式中，头部有巨大的标志结构，数据稀疏度达到了 98%。而 bin 格式则无稀疏数据。

```
nasm -f elf boot.s -o boot.o
352-bytes-header omitted
fafa 668c c88e d88e c0e8 0e00 0000 e82c
0000 00b0 43e8 3e00 0000 ebfe b000 5554
6650 6651 6652 b406 66b9 0000 8a35 9f00
0000 b250 cd10 665a 6659 6658 5c5d c366
ba00 0055 5466 5066 53b4 02b7 00cd 1066
5b66 585c 5dc3 b041 5554 6650 6653 6651
b409 b700 b30f 66b9 0100 cd10 6659 665b
```

图 2.5，ELF 格式

```
bin: nasm -f bin boot.s -o boot.o
fafa 8cc8 8ed8 8ec0 e80a 00e8 2500 b043
e838 00eb feb0 0066 5566 5450 5152 b406
b900 008a 3685 00b2 50cd 105a 5958 665c
665d c3ba 0000 6655 6654 5053 b402 b700
cd10 5b58 665c 665d c3b0 4166 5566 5450
5351 b409 b700 b30f b901 00cd 1059 5b58
665c 665d c350 80fa 5072 07b2 00fe c6e9
0200 fec2 3a36 8500 7609 b001 e898 ff8a
```

图 2.5，BIN 格式

像上图 2.6 这样的结构我们也应该按照 ABI 来剪裁，本次研究中使用的机器是 darwin Mach-O 格式，该格式与 linux 系统采用的 ELF（Executable and Linkable Format）格式类似，如图 2.7 所示。

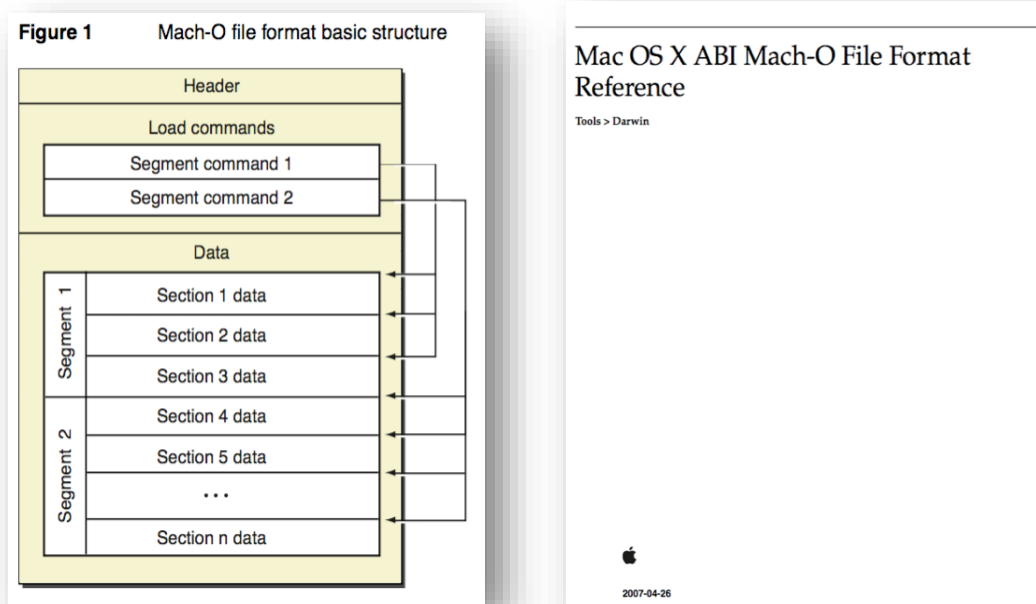


图 2.7，Mach-O 格式

通过阅读苹果的 ABI 说明书，我们编写了工具解析该格式，以达到定位首函数、定位中端服务函数并裁减二进制文件的目的。最后我们遇到了 opcode 问题，为了代码的清晰、结构的明确，我们需要链接多个文件，bin 格式虽然紧凑但无法链接，所以不能使用 bin 格式，而在使用其他可链接格式的过程中，gcc 与 unix-cc 编译 C 语言代码时虽然指定了相应的参数，生成了 i386 代码，但是却无法在机器上正确运行。我们通过反汇编发现，指令中立即数部分被延长了，如图 2.8 所示。

```

00007eb0: cli                                ; fa
00007eb1: cli                                ; fa
00007eb2: mov ax, cs                        ; 668cc8
00007eb5: mov ds, ax                        ; 8ed8
00007eb7: mov es, ax                        ; 8ec0
00007eb9: call .+14                         ; e80e00
00007ebc: add byte ptr ds:[bx+si], al ; 0000
00007ebe: call .+44                         ; e82c00
00007ec1: add byte ptr ds:[bx+si], al ; 0000
00007ec3: mov al, 0x43                      ; b043
00007ec5: call .+62                         ; e83e00
00007ec8: add byte ptr ds:[bx+si], al ; 0000
00007eca: jmp .-2                           ; ebfe

```

图 2.8，反汇编的结果

因为 intel 指令集是变长指令，上述地址的翻译错误导致了后续代码空间顺序紊乱。我们阅读 gcc 与 cc 的说明书后，根据其声称的编写 boot 代码时所需设置再次进行试验，发现没有任何效果，代码依然不是 i386-generic，而是

i386-long 模式。经过一系列的失败和探索，我们发现只能将编译过程手动分解为前端、汇编、链接、后处理，四步操作，并在其中采用大量的参数、配合自写脚本，才能使得 cc\gas\ld 输出正确的结果。我们最终探索得到的步骤可在 Makefile 中体现。简而言之抽取其中四条命令和部分参数，其步骤可大致理解为：

1. nasm -f bin
2. cc -m16 \$(csrc) -S -O0 -fPIC -ffreestanding
3. Asm(“.code16 \n\t”);
4. as -arch i386 \$(target).s -o \$(target).o -O0

编译的过程输出如图 2.9 所示。

```
chens-MacBook:asm chen$ make boot target=syscall img=test count=8
cc -m16 syscall.c -S -O0 -fPIC -ffreestanding
as -arch i386 syscall.s -o syscall.o -O0
cp syscall.o syscall_mid.o
otool -l syscall_mid.o > tmp.txt
dd if=syscall_mid.o of=syscall.o isseek=`python strip.py tmp.txt` bs=1 count=3000
3000+0 records in
3000+0 records out
3000 bytes transferred in 0.009422 secs (318410 bytes/sec)
rm tmp.txt
locating int08 function entry...
-----
00000330 <_int08>:
-----
dd if=syscall.o of=test.img bs=512 count=8 conv=notrunc
5+1 records in
5+1 records out
3000 bytes transferred in 0.000034 secs (88612056 bytes/sec)
build success
chens-MacBook:asm chen$
```

图 2.9，编译过程

相比之前几次探究，这次 bootloader 的主要改变是放弃劫持 lch，转而劫持其父硬件中断 08h，绑定到 mini-kernel 中的 int08 函数上。代码如下。

```
16 ;; set interrupt
17 mov al,0x04
18 mov ah,0x08 ;int number
19 mul ah ; ax=08h*4
20 ;lea dx,[foo]
21 mov dx,0x0330 ;assign IP
22 mov di,ax ;di=8*4
23 mov [di],dx ;put foo IP
24 add di,2
25 mov ax,0x07e0 ;assign CS
26 mov [di],ax ;put foo CS
```

另一处不同，是对堆栈的处理，之前我们一直让堆栈处于 BIOS 执行完的默认状态，但是我们发现，如果在远跳转后依然保留堆栈为 0000:ffff 尺度的话，会因为字符串类操作与 c 语言传参操作占堆栈大而溢出到 0x7e00 以上的代码空间，所以我们在这里重新设置了对栈的大小，代码如下。

```
28 mov esi,0
29 mov edi,0
30
31 mov ax,0x07e0
32 mov ss,ax
33 mov esp,0xff00
34 mov ebp,esp
35 jmp dword 0x07e0:0x0000
```

下面将介绍说明 C 语言实现的 mini-kernel。首先是文件头，如下所示。

```
1  asm(".code16 \n\t");
2  /*** INCLUDE AND DEFINE ***/
3  #define TRUE 1
4  #define FALSE 0
5  #define POOL_SIZE 2
6  #define TTY_ROWS 25
7  #define TTY_COLS 80
8  #define NULL 0
9  #define UNDEFINED 0
10 #define READY 1
11 #define RUNNING 2
12 #define PAUSED 3
```

这里定义了 C 语言一些默认的符号，定义了进程池的大小，TTY 尺寸，与进程的状态标志。之后我们定义了进程描述符，结构如下所示。

```
18 typedef struct task_desc
19 {
20     byte tid;
21     byte state; // 1 running; 0 not running
22     uint16 flags;
23     uint16 cs;
24     uint16 ip;
25     uint32 stack_base_ebp;
26     uint32 ebp;
27     uint32 esp;
28     uint32 eax;
29     uint32 ebx;
30     uint32 ecx;
31     uint32 edx;
32     uint32 esi;
33     uint32 edi;
34     void (* procedure)();
35     uint32 bar;
36     byte stack[128];
37 } TD;
```

进程描述符内主要存储了寄存器信息、id、进程状态、进程主函数与堆栈保留空间。Bar 变量的作用是检测与防止堆栈读写问题导致溢出，使得系统能对该异常做出反应。

主要的示例过程与工具函数如下所示。

```
39  /*** FUNCTION DECLARATION  ***/
40  void int08(void);
41  void roottask(void);
42  void task1(void);
43  void clear(void);
44  void setcursor(byte row,byte col);
45  int len(char *strp);
46  void dispchar(char c);
47  void s_dispchar(char c);
48  void print(char *p);
49  void sleep(int ms);
50  uint32 getesp(void);
51  void start_procedure(byte tid ,byte sti);
52  void new_task(void (*procedure)(), byte tid);
53  void switch_proc(void);
54  void s_setcursor(byte row, byte col);
```

功能分别为：中断伺服、系统 idle 零号进程、客户进程、清屏调用、光标调用、字符串求长、非安全栈显示字符、安全显示字符、打印字符串、自旋睡眠、获得即时 esp、开始调度 API、加入任务、任务切换、非栈安全设置光标。除此外还有许多辅助函数，细节繁杂就不一一解释了，只挑出其中比较重要的任务切换相关代码说明。

```
64  void main(void)
65  {
66      asm("pushl %eax \n\t"
67          "movw %cs,%ax \n\t"
68          "movw %ax,%ds \n\t"
69          "movw %ax,%es \n\t"
70          "movw %ax,%fs \n\t"
71          "movw %ax,%gs \n\t"
72          "movw %bp,%di \n\t"
73          "popl %eax \n\t"
74          );
75      setcursor(1,0);
76      new_task(roottask,0);
77      new_task(task1,1);
78      start_procedure(0,FALSE);
79  }
80  }
```

主函数设置好段，清理堆栈，然后加入两个进程开始调度。如以上的代码。

```

82 void int08(void)
83 {
84     asm("pushl %eax \n"
85         "pushal \n"
86         "movl %ebp,%eax \n"
87         "addl $10,%eax \n"
88         "movl %eax,12(%esp) \n" //old esp
89         "movl (%ebp),%eax \n" //get old ebp
90         "movl %eax,8(%esp) \n" //save old ebp
91         "movw 8(%ebp),%ax \n" //old flags
92         "pushw $0x0 \n"
93         "pushw %ax \n"
94         "movw 6(%ebp),%ax \n" //old cs
95         "pushw $0x0 \n"
96         "pushw %ax \n" //save cs
97         "movw 4(%ebp),%ax \n" //old ip
98         "pushw $0x0 \n"
99         "pushw %ax \n" //save old ip
100        "calll _savecontext \n"
101        "addl $44,%esp \n"
102        "popl %eax \n"
103        );
104    switch_proc();
105 }

```

以上代码是中断处理函数，首先将当前的通用寄存器都压栈，再将堆栈指针都压栈，然后压入 cs、ip 后调用保存上下文函数，上下文保存完毕后，清理堆栈保证无任何泄漏，再进入进程切换。下面首先展示上下文保护函数，如下。

```

121 void savecontext(uint16 ip,
122 uint16 cs,
123 uint16 flags,
124 uint32 edi,
125 uint32 esi,
126 uint32 ebp, // ebp when interrupted
127 uint32 esp, // esp when interrupted
128 uint32 ebx,
129 uint32 edx,
130 uint32 ecx,
131 uint32 eax)
132 {
133     uint32 stack_usage=0 ;
134     TD * t = &(pool[current_task]) ;
135     byte * ss = t->stack;
136     t->ip = ip ;
137     t->cs = cs ;
138     t->flags = flags ;
139     t->edi = edi ;
140     t->esi = esi ;
141     t->ebp = ebp ;
142     t->esp = esp ;
143     t->ebx = ebx ;
144     t->edx = edx ;
145     t->ecx = ecx ;
146     t->eax = eax ;
147
148     stack_usage = (t->stack_base_ebp) - (t->esp);
149     stack_usage+=3; //to save 4B data ahead-of task-ebp
150     if(stack_usage>=128)
151     {
152         stack_usage=128;
153     }
154     asm("cld \n"
155         "rep movsb \n"
156         :
157         : "c"(stack_usage), "S"(esp), "D"(ss)
158         );
159 }
160

```

该函数依据 gcc 说明书所描述的 C 语言标准传参过程，从右到左依次弹出，获得参数，保存到任务描述符中，并调用 MOVSB 指令保存当前任务从栈底到栈顶的所有信息到任务描述符中。

该函数执行完毕后，int08 将会调用 switch_proc 函数。该函数上半部分形式如下面的代码。

```
162 void switch_proc(void)
163 {
164     if(pool[1].state==READY)
165     {
166
167         asm("pushl %eax \n"
168             "movw $0,%ax \n"
169             "pushw %ax \n"
170             "popfw \n"
171             "movb $0x20,%al \n"
172             "outb %al,$0x20 \n"
173             "popl %eax \n"
174             );
175         start_procedure(1,TRUE); // no return;
176
177     }
178
179     pool[current_task].state=PAUSED;
180     if(current_task==0)
181         current_task=1;
182     else
183         current_task=0;
184     pool[current_task].state=RUNNING;
185
186
187     TD * t = &(pool[current_task]);
188     uint32 stack_usage = (t->stack_base_ebp) - (t->esp) + 3;
189     byte * ss = t->stack;
190     if(stack_usage>=128)
191         stack_usage=128;
```

首先 switch_proc 会选择一个程序来调度，在此处可以实现任意的调度算法或逻辑，本次实验中采取了便于解释 2 个进程关系的处理方式，即：只有 idle 进程时启动另外一个 ready 进程，idle 进入 paused 状态，当两个进程都已启动时，进行公平时间片调度，被切换掉的进程设置为 paused，新运行的进程设置为 running。

紧接着是通用的处理过程，主要关注如何恢复现场。首先检测被还原的进程所占堆栈式否超过了 128B 上限，如果超出了就截断，否则继续。这是为了防止客户在编写程序时出现错误而无限地毁坏堆栈，伤及代码；另一方面也是为了限制管理开销。同时，我们也可以看到，为了防止调度嵌套，我们只在控制转移之前向 20h 端口发送“看门狗”喂狗信号，允许再次产生中断，这个做法在后面一半代码中也有体现。

接下来展示的是 switch_proc 的后半部分代码。

```
193     asm("movw $0x00,-2(%%esi) \n" // ip
194         "movw %%ax,-4(%%esi) \n" // ip
195         "movw %%bx,-6(%%esi) \n" //flags
196         "movl %%ecx,-10(%%esi) \n" //edi
197         "movl %%edx,-14(%%esi) \n" //esi
198         :
199         : "S"(t->esp), "a"(t->ip), "b"(t->flags), "c"(t->edi), "d"(t->esi)
200         );
201
202     asm("movl %%eax,-18(%%esi) \n" //eax
203         "movl %%ebx,-22(%%esi) \n" //ebx
204         "movl %%ecx,-26(%%esi) \n" //ecx
205         "movl %%edx,-30(%%esi) \n" //edx
206         "movl %%esi,%%edx \n" // save esp in edx
207         :
208         : "a"(t->eax), "b"(t->ebx), "c"(t->ecx), "d"(t->edx)
209         );
210
211     asm("\n"
212         :
213         : "b"(t->ebp) // save ebp in ebx
214         );
215     // recover lower stack
216
217     asm("cld \n"
218         "rep movsb \n"
219         :
220         : "c"(stack_usage), "S"(ss), "D"(t->esp)
221         );
222
223     asm("movl %%ebx,%%ebp \n"
224         "movl %%edx,%%esp \n"
225         "movb $0x20,%al \n"
226         "outb %al,$0x20 \n"
227         "subl $30,%%esp \n"
228         "popl %%edx \n"
229         "popl %%ecx \n"
230         "popl %%ebx \n"
231         "popl %%eax \n"
232         "popl %%esi \n"
233         "popl %%edi \n"
234         "popfw \n"
235         "sti \n"
236         "retl \n"
237         );
238 }
```

这部分代码的作用是将保存在任务描述符中的寄存器、堆栈指针恢复到对应硬件，再将堆栈写回内存，最后恢复标志位寄存器、恢复 cs、ip。

因为我们要恢复所有寄存器，而总是需要额外寄存器作为中转器，这就像玩华容道一样比较难搞，所以我们将要恢复的内容依次写到堆栈里，然后逐个弹出。注意，这个堆栈的位置不是随意指定的，否则客户程序栈内存恢复的过程中会破坏我们的寄存器临时栈，所以我们将这个位置显式地写在旧 esp 的后面。旧的 esp 存在 esi 中。

首先，我们利用嵌入式汇编从任务描述符中恢复 ip、flags、edi、esi，然后恢复 eax/ebx/ecx/edx，然后取出 ebp。此时寄存器都已经在 esp 栈顶存放了，我们依次将数据复制到旧的程序栈中。接着，将各个寄存器的值通过 pop 重新赋给寄存器，完成恢复。

在喂“看门狗”开中断后，我们将返回 ip、cs、flag 写到栈顶，利用正常的 retl 跳转到原程序运行的位置，至此完成调度。这种完全保存栈行为的切换方式能够调度任何程序，包括线性时无关或任意时相关程序（也可理解为可重入、不可重入程序）。此处有趣的一点是，int08 函数进入 switch_proc 函数后是不会返回的，int08 函数也不会返回，因为我们模拟了正常时间中断的行为后，伪造了一个正常函数调用的现场利用 retl 跳转到其他位置去了。

Idle 程序打印字母“I”，task 程序打印字母“O”，实现效果如图 2.10 所示。

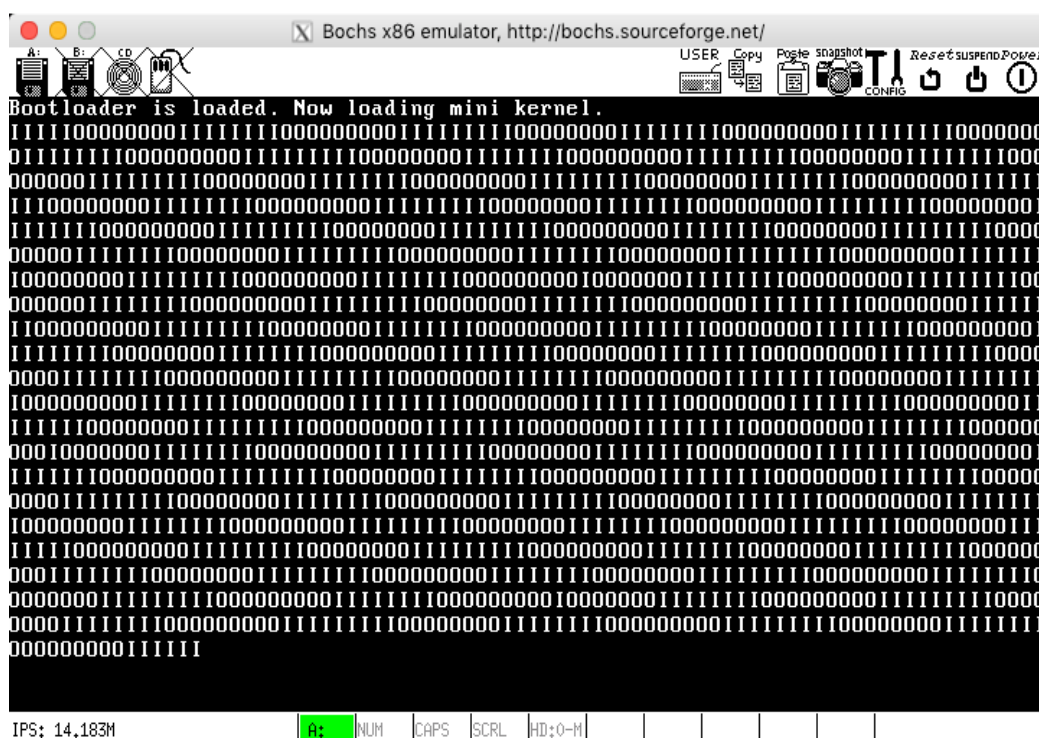


图 2.10，实现效果

成功进行了 bootloader 加载、minikernel 加载、调度、保存、切换等一系列事件驱动调度器的功能。

第三章 保护模式

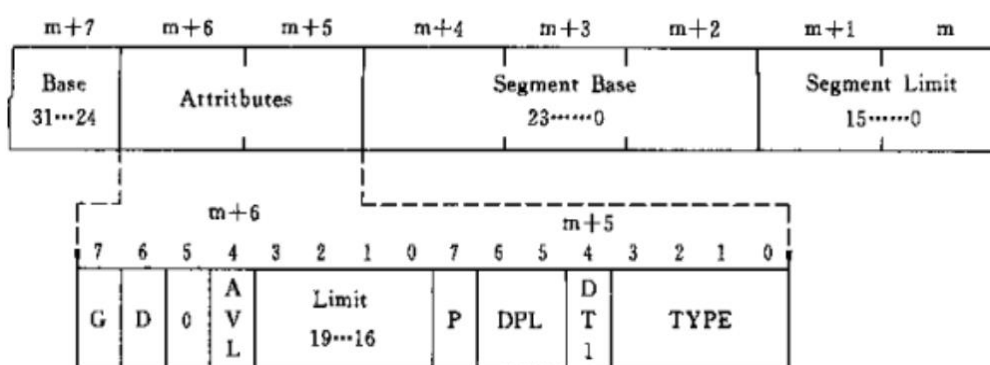
32 位数据线和 32 位地址线，支持存储器分段和分页管理，支持多任务，可以进行快速的任务切换和保护任务环境，在分页的基础上实现了虚拟存储器，包含 4 个特权级和完善的特权检查机制。

描述符可用于描述多种对象：存储段、任务状态段、调用门、任务门、中断门、陷阱门和 LDT；80386 将描述符组织成线性表，称为描述符表，每个描述符表本身形成一个特殊的内存数据段，最多可以包含 8192 个描述符，该段由操作系统维护、由处理器中的存储管理单元硬件 MMU 访问；整个系统中全局描述符 GDT 和中断描述符表 IDT 只有一张，局部描述符表 LDT 可以有若干张，每个任务一张。

GDT：含有操作系统所使用的代码段、数据段和堆栈段的描述符，在任务切换时，并不切换 GDT，通过 GDT 可以使各任务都需要的段能够被共享；LDT：每个任务的局部描述符表 LDT 都含有该任务自己的代码段、数据段和数据段的描述符。也包含该任务使用的一些门描述符。随着任务的切换，系统当前的局部描述符 LDT 也随之切换。

3.1 描述符 Descriptor

- 存储段描述符：代码段、数据段、堆栈段；系统描述符：任务状态段 TSS、局部描述符表 LDT



P 存在位：1 代表描述符所描述的段在内存中；0 该描述符进行内存访问会引起异常。

DPL 描述符特权级，2 位，规定了所描述的段特权级，用于特权检查，决定该段可否访问。

DT 描述符类型：1 代表存储段描述符，0 代表系统段描述符和门描述符。

TYPE：说明存储段描述符所描述的存储段的具体属性：

数据段类型	类型值说明	代码段类型
-----		-----
0	只读	8 只执行
1	只读、已访问	9 只执行、已访问
2	读/写	A 执行/读
3	读/写、已访问	B 执行/读、已访问
4	只读、向下扩展	C 只执行、一致码段
5	只读、向下扩展、已访问	D 只执行、一致码段、已访问
6	读/写、向下扩展	E 执行/读、一致码段
7	读/写、向下扩展、已访问	F 执行/读、一致码段、已访问

系统段类型 类型编码说明

0 <未定义>
1 可用286TSS
2 LDT
3 忙的286TSS
4 286调用门
5 任务门
6 286中断门
7 286陷阱门
8 未定义
9 可用386TSS
A <未定义>
B 忙的386TSS
C 386调用门
D <未定义>
E 386中断门
F 386陷阱门

G 段界限粒度：针对段界限有效，段基址总是以字节为单位（无效）；0 代表粒度为字节，1 代表粒度为 4K 字节。

D 位可用于描述可执行段、向下扩展数据段、或由 SS 寄存器寻址的段（通常是堆栈段）。在描述可执行段的描述符中，D 位决定了指令使用的地址及操作数所默认的大小。

① D=1 表示默认情况下指令使用 32 位地址及 32 位或 8 位操作数，这样的代码段也称为 32 位代码段；

② D=0 表示默认情况下，使用 16 位地址及 16 位或 8 位操作数，这样的代码段也称为 16 位代码段，它与 80286 兼容。可以使用地址大小前缀和操作数大小前缀分别改变默认的地址或操作数的大小。

在向下扩展数据段的描述符中，D 位决定段的上部边界。

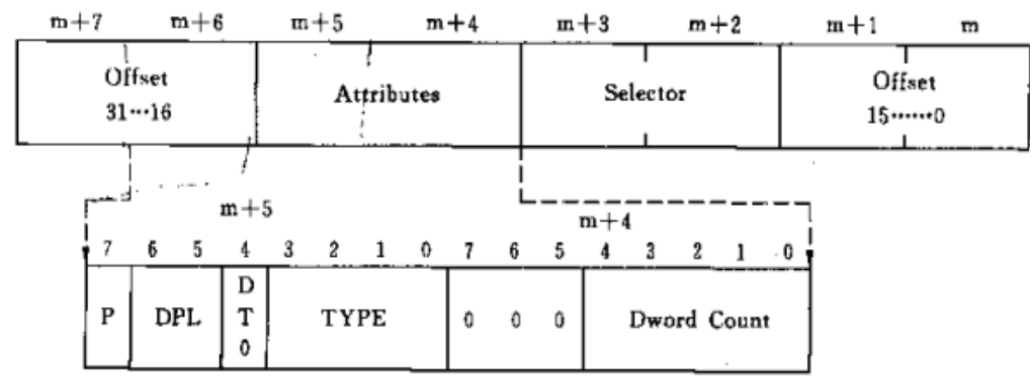
- ② D=1 表示段的上部界限为 4G；
- ② D=0 表示段的上部界限为 64K，这是为了与 80286 兼容。

在描述由 SS 寄存器寻址的段描述符中，D 位决定隐式的堆栈访问指令(如 PUSH 和 POP 指令)使用何种堆栈指针寄存器。

- ① D=1 表示使用 32 位堆栈指针寄存器 ESP；
- ② D=0 表示使用 16 位堆栈指针寄存器 SP，这与 80286 兼容。

AVL 软件可利用位：Undefined

• 门描述符



Dword Count: 从调用者堆栈中将参数复制到被调用者堆栈中，参数个数由调用门中该项指定，为 0 代表不会复制参数包括中断门，调用门，陷阱门，任务们。

3.2 选择子 Selector

• 结构

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
描述符索引													TI	RPL	

• RPL(Requested Privilege Level):请求特权级，用于特权检查
 TI(Table Indicator):引用描述符表指示位：0 代表从全局描述符表中读取描述符，1 代表从局部描述符表 LDT 中读取描述符索引为对应描述符表中的偏移量。

3.3 80386 寄存器结构

4 个 32 位通用寄存器：EAX，EBX，ECX，EDX；4 个 32 位地址寄存器 ESP，EBP，EDI，ESI；32 位指令指针寄存器 EIP；32 位标志寄存器 EFALGS；6 个 16 位段寄存器 CS，DS，ES，SS，FS，GS；4 个 32 位控制寄存器 CR0，CR1，CR2，CR3；4 个系统地址寄存器：GDTR（48），IDTR（48），LDTR（16），TR（16）。

	31	16	15	87	0		31	16	15	0
EAX	(AH)AX(AL)					ESP	SP			
EBX	(BH)BX(BL)					EBP	BP			
ECX	(CH)CX(CL)					ESI	SI			
EDX	(DH)DX(DL)					EDI	DI			

15	0	31	0	19	0	11	0
CS 选择器		CS 描述符高速缓存寄存器					
SS 选择器		SS 描述符高速缓存寄存器					
DS 选择器		DS 描述符高速缓存寄存器					
ES 选择器		ES 描述符高速缓存寄存器					
FS 选择器		FS 描述符高速缓存寄存器					
GS 选择器		GS 描述符高速缓存寄存器					

	31	16	15	0		31	16	15	0
EIP	IP				EFLAGS	FLAGS			

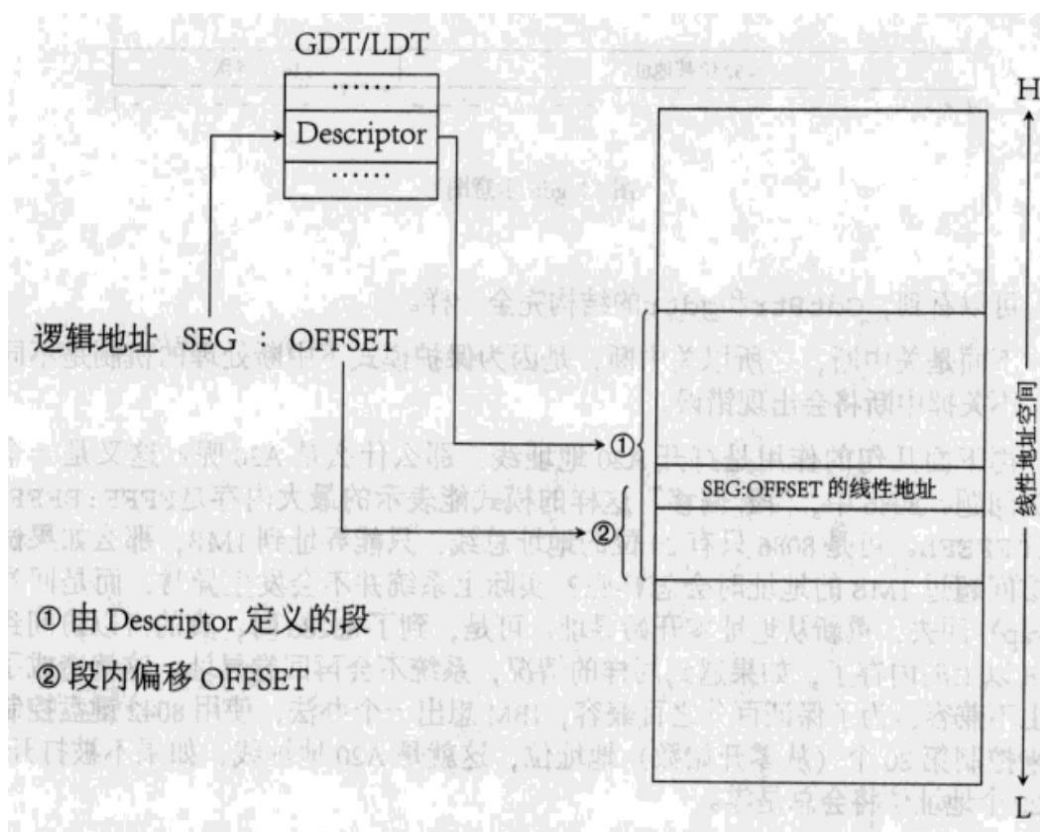
	31	0	15	0
GDTR				
IDTR				

	15	0	31	0	19	0	11	0
LDTR	LDTR 选择器		LDTR 高速缓存					
TR	TR 选择器		TR 高速缓存					

GDT 和 IDT 的基地址分别存放在 GDTR 和 IDTR 中，而各个 LDT 的基地址存放在 GDT 的 LDT 描述符中，LDTR 是当前所使用的 LDT 描述符的选择子；GDTR 和 IDTR 长 48 位，高 32 位为基地址，低 16 位为界限；任务状态段寄存器 TR 包含指示描述当前任务的任务状态段的描述符的选择子，从而规定了当前任务的状态段。

3.4 保护模式下的寻址

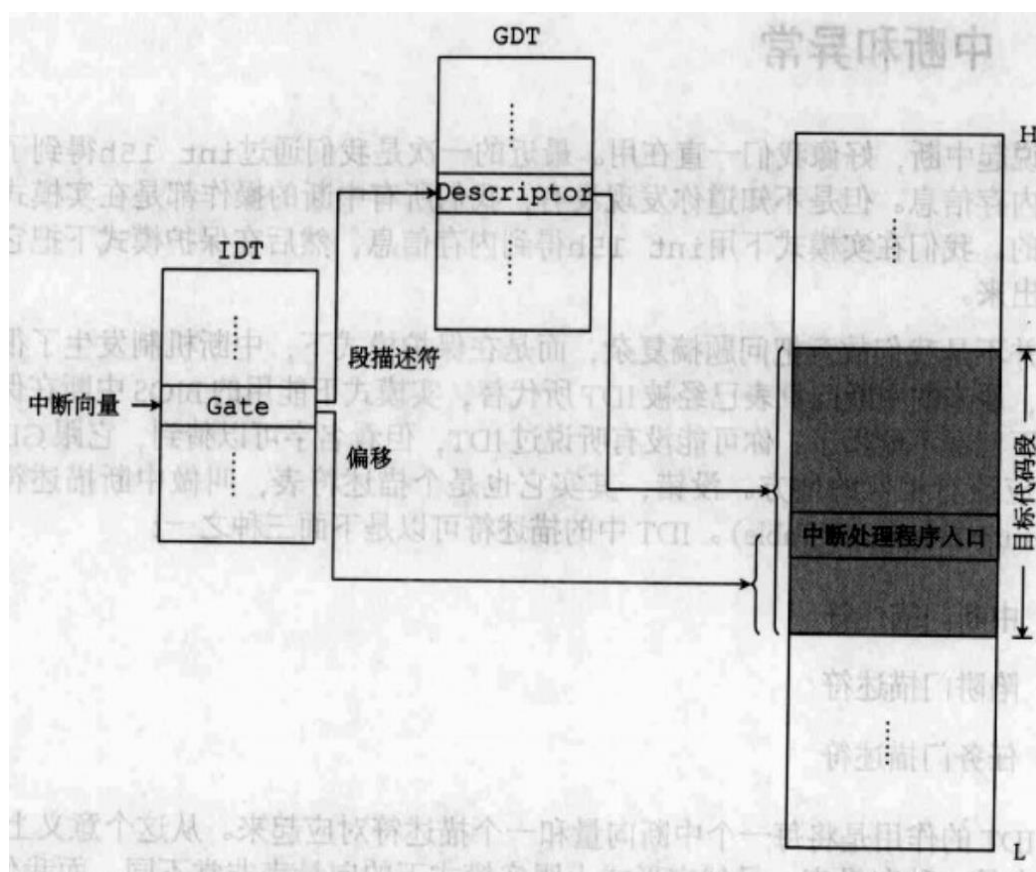
段选择子装入 CS，CS 描述符高速缓存寄存器装入代码段描述符，包括首地址、界限、属性（选择子装入段寄存器选择器，在 GDT 中选择对应段描述符，将该描述符装入对应段寄存器的描述符高速缓存寄存器）。



指令执行过程

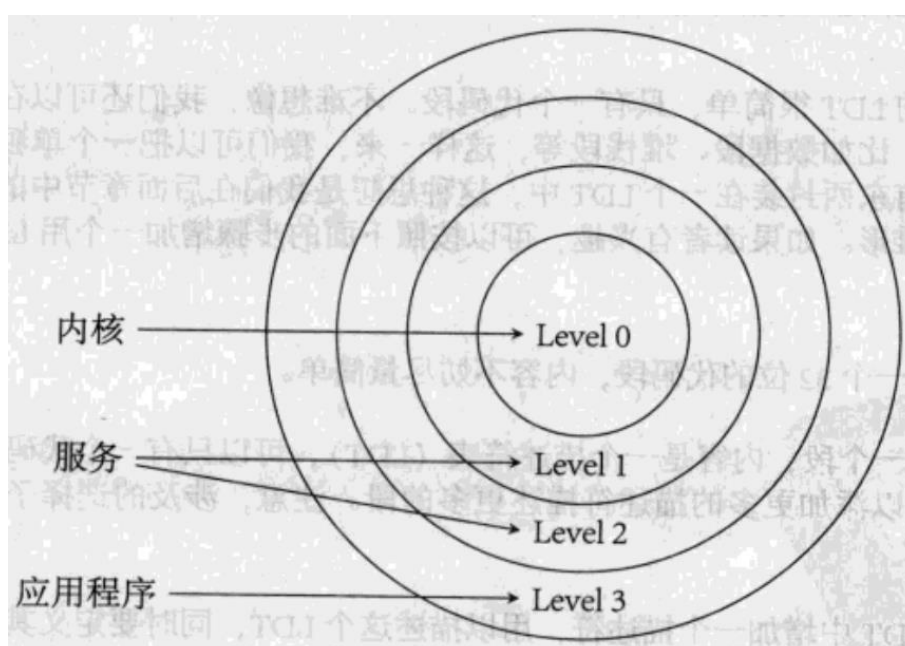
- ① CS 描述符高速缓存寄存器中基地址+EIP（偏移），形成 32 位地址，到此地址取得命令。
- ② EIP=EIP+所指指令的长度。
- ③ 执行指令（个别指令执行的时候修改 CS 描述符高速缓存寄存器, EIP），跳到①继续执行。

- 中断寻址



3.5 特权权

- 4 级特权



- 特权级检验

CPL：当前执行的程序或任务的特权级，存储在 CS 和 SS 的第 0 位和第一位；通常情况下，CPL 等于代码所在的段的特权级，当程序进行转移到不同特权级的代码段时，处理器改变 CPL（当处理器访问一个与 CPL 特权级不同的一致代码段时，CPL 不变）。

DPL：段或者门特权级。数据段的 DPL 规定了可以访问此段的最低特权级；非一致代码段（不使用调用门）的 DPL 规定访问此段的特权级，CPL 必须等于目标访问段的 DPL，RPL 小于 DPL；调用门的 DPL 规定了当前执行的程序或任务可以访问此调用门的最低特权级；一致代码段和通过调用门访问的非一致代码段 DPL 规定了访问此段的最高的特权级；TSS 的 DPL 规定了可以访问此 TSS 的最低特权级。

RPL：段选择子的第 0 位和第 1 位。处理器通过检查 RPL 和 CPL，同 DPL 进行比较，来确认一个访问段或门的请求是否合法，其中 RPL 较 CPL 起决定作用

- 不同特权级代码段之间的转移：从一个代码段转移到另一个代码段之前，处理器检查描述符的界限、类型、特权级等，检验通过后将目标代码段的选择子加载到 CS 中；转移通过跳转指令或中断或异常引起。

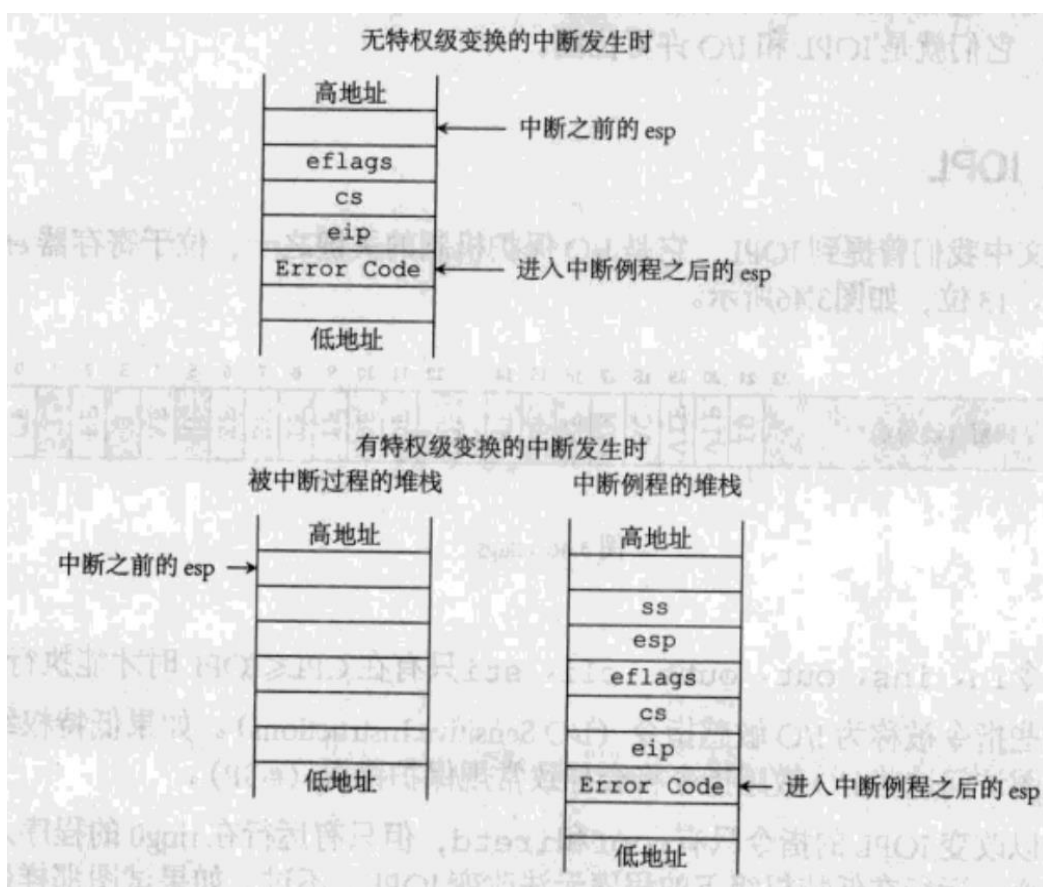
- 使用调用门来实现不同特权级代码之间的转移。

- 特权级发生变换时相应的堆栈也要发生变化，处理器的这种机制避免了高特权级的过程由于栈空间不足而崩溃，另外不同特权级共享一个堆栈的话，高特权级的程序也有可能受到干扰。

Jmp 指令直接跳转到目的地址，不影响堆栈；Call 指令分长调用和短调用，会影响堆栈。要注意这两种指令长跳转和短跳转时对段寄存器和 EIP（偏移量）的影响。

call 调用门也是一个长调用，但是涉及堆栈切换时，堆栈已经发生变化，intel 将堆栈 A 的诸多内容复制到堆栈 B（参数、调用者 SS、ESP、CS、EIP），当需要获取不同堆栈的 SS 和 ESP 时，使用 TSS 的数据结构。

- 中断或异常发生时的堆栈变化



如果有出错码的话，出错码在最后压栈；从中断返回必须使用 `iretd`，返回时同时改变 `eflags`。

3.6 2 级加载的保护模式下的调度器

`pm.inc` 是宏定义文件，用于在 `boot.s` 中定义描述符和门描述符，因此在 `boot.s` 中直接使用宏定义对要使用的描述符和门描述符进行定义。`Boot.s` 汇编语言程序文件，是在保护模式下实现的两个进程间的调度。这两个进程是用 `fen` 别在屏幕上打印 A 字符和 B 字符，进程切换的时候，屏幕上打印不同的字符。

`pm.inc` 代码如下：

```

;-----
; 描述符类型值说明
; 其中：
;     DA_ : Descriptor Attribute
;     D   : 数据段
;     C   : 代码段
;     S   : 系统段
;     R   : 只读
;     RW  : 读写

```

```

;      A      : 已访问
;      其它    : 可按照字面意思理解
;      G D 0 AVL 0 0 0 0 P DPL(2 位) DT TYPE(4 位)
;-----
DA_32      EQU      4000h ; 32 位段 0100 0000 0000 0000

DA_DPL0     EQU      00h  ; DPL = 0 0000 0000
DA_DPL1     EQU      20h  ; DPL = 1 0010 0000
DA_DPL2     EQU      40h  ; DPL = 2 0100 0000
DA_DPL3     EQU      60h  ; DPL = 3 0110 0000
;-----
; 存储段描述符类型值说明
;-----
DA_DR       EQU      90h  ; 存在的只读数据段类型值 1001 0000
DA_DRW      EQU      92h  ; 存在的可读写数据段属性值 1001 0010
DA_DRWA     EQU      93h  ; 存在的已访问可读写数据段类型值 1001 0011
DA_C        EQU      98h  ; 存在的只执行代码段属性值 1001 1000
DA_CR       EQU      9Ah  ; 存在的可执行可读代码段属性值 1001 1010
DA_CCO      EQU      9Ch  ; 存在的只执行一致代码段属性值 1001 1100
DA_CCOR     EQU      9Eh  ; 存在的可执行可读一致代码段属性值 1001 1110
;-----
; 系统段描述符类型值说明
;-----
DA_LDT      EQU      82h  ; 局部描述符表段类型值 1000 0010
DA_TaskGate EQU      85h  ; 任务门类型值 1000 0101
DA_386TSS   EQU      89h  ; 可用 386 任务状态段类型值 1000 1001
DA_386CGate EQU      8Ch  ; 386 调用门类型值 1000 1100
DA_386IGate EQU      8Eh  ; 386 中断门类型值 1000 1110
DA_386TGate EQU      8Fh  ; 386 陷阱门类型值 1000 1111
;-----

;-----
; 选择子类型值说明
; 其中:
;      SA_    : Selector Attribute

SA_RPL0     EQU      0      ; 00
SA_RPL1     EQU      1      ; 01
SA_RPL2     EQU      2      ; 10
SA_RPL3     EQU      3      ; 11

SA_TIG      EQU      0      ; 0000
SA_TIL      EQU      4      ; 0100
;-----
; 宏 -----
;-----
;
; 描述符
; usage: Descriptor Base, Limit, Attr
;      Base: dd

```

```

;      Limit: dd (low 20 bits available)低二十位可用
;      Attr:  dw (lower 4 bits of higher byte are always 0)高字节的低四位始终为0
%macro Descriptor 3 ;段界限为低地址 1 代表 Base 2 代表 Limit 3 代表属性
    dw      %2 & 0FFFFh                ; 段界限 1    (2 字节)
    dw      %1 & 0FFFFh                ; 段首地址 1   (2 字节)
    db      (%1 >> 16) & 0FFh          ; 段首地址 2   (1 字节)
    dw      ((%2 >> 8) & 0F00h) | (%3 & 0F0FFh) ; 属性 1 + 段界限 2 + 属性 2
; (2 字节)
    db      (%1 >> 24) & 0FFh          ; 段首地址 3   (1 字节)
%endmacro ; 共 8 字节
;
; 门
; usage: Gate Selector, Offset, DCount, Attr
;      Selector:  dw
;      Offset:    dd
;      DCount:    db
;      Attr:      db
%macro Gate 4 ;1 代表 Selector 2 代表 Offset 3 代表 DCount 4 代表 Attr
    dw      (%2 & 0FFFFh)              ; 偏移 1      (2 字节)
    dw      %1                          ; 选择子      (2 字节)
    dw      (%3 & 1Fh) | ((%4 << 8) & 0FF00h) ; 属性        (2 字节)
    dw      ((%2 >> 16) & 0FFFFh)        ; 偏移 2      (2 字节)
%endmacro ; 共 8 字节

```

在 boot.s 中的 GDT 和 IDT 代码如下

```

#include "pm.inc"

org 07e00h
jmp LABEL_BEGIN

[SECTION .gdt]
; GDT
; 0x7c04      段基址,      段界限,      属性
LABEL_GDT:    Descriptor 0,      0,      0
LABEL_DESC_CODE32: Descriptor 0, SegCode32Len - 1, DA_C + DA_32
LABEL_DESC_VIDEO: Descriptor 0B8000h,      0ffffh, DA_DRW
; GDT finished

GdtLen        equ  $ - LABEL_GDT
GdtPtr        dw   GdtLen - 1
              dd   0

; GDT Selector
SelectorCode32 equ LABEL_DESC_CODE32 - LABEL_GDT
SelectorVideo  equ LABEL_DESC_VIDEO - LABEL_GDT
; END of [SECTION .gdt]

; IDT
[SECTION .idt]
ALIGN 32
[BITS 32]
LABEL_IDT: ; 0x00007c40
.00h:      Gate SelectorCode32,      ClockHandler,      0, DA_
386IGate
.01h:      Gate SelectorCode32,      UserIntHandler,      0, DA_
386IGate

IdtLen        equ  $ - LABEL_IDT
IdtPtr        dw   IdtLen - 1
              dd   0
; END of [SECTION .idt]

```

boot.s 进入保护模式的实现代码：

```
    ; 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GDT作准备
    xor    eax, eax
    mov    ax, ds
    shl    eax, 4
    add    eax, LABEL_GDT
    mov    dword [GdtPtr + 2], eax
    ; 为加载IDT作准备
    xor    eax, eax
    mov    ax, ds
    shl    eax, 4
    add    eax, LABEL_IDT
    mov    dword [IdtPtr + 2], eax

    lgdt   [GdtPtr]

    cli

    lidt   [IdtPtr]

    ; 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
; END of [SECTION .s16]
```

结果显示如图 2.2。

第四章 总结

本次实验已经成功地，基于 Bochs 在两个不同模式下分别是 8086 的实模式和 80386 的保护模式下，实现了两个简单任务的调度器。在实现过程中有对比较底层的内核结构和使用进行详细的描述。因为本次实验对编译的需求比较复杂，所以我们除了实现汇编的编写还有编写了 C 语言，两个语言结合地使用就是为了读者更加容易理解。

参考文献

- [X86 WIKIBOOK] https://en.wikibooks.org/wiki/X86_Assembly
- [IntelAsm] <http://www.logix.cz/michal/doc/i386>
- [NASM 备忘 lmu] <http://cs.lmu.edu/~ray/notes/nasmtutorial/>
- [Nasm Manual1] <http://www.nasm.us/doc/nasmdoc0.html>
- [NASM 数据定义] <http://www.nasm.us/doc/nasmdoc3.html>
- [NASM] https://www.tutorialspoint.com/assembly_programming/assembly_system_calls.htm
- [GAS] <http://csiflabs.cs.ucdavis.edu/~ssdavis/50/att-syntax.htm>
- [Gas Doc] <https://sourceware.org/binutils/docs-2.16/as/index.html>
- [macOS] <http://orangejuiceliberationfront.com/intel-assembler-on-mac-os-x/>
- [INT table] https://en.wikibooks.org/wiki/X86_Assembly/Advanced_Interrupts
- [Inline Assembly] <http://www.ibm.com/developerworks/library/l-ia/index.html>
- [InlineAssemble] [<http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html>]
- [Code Optimization] <http://www.agner.org/optimize/>
- [Apple ABI Function Call Guide] [[link](#)]