

# 操作系统实验一 调试分析 Linux 0.00 引导程序

## 一、实验目的

- 熟悉实验环境；
- 掌握如何手写 Bochs 虚拟机的配置文件；
- 掌握 Bochs 虚拟机的调试技巧；
- 掌握操作系统启动的步骤。

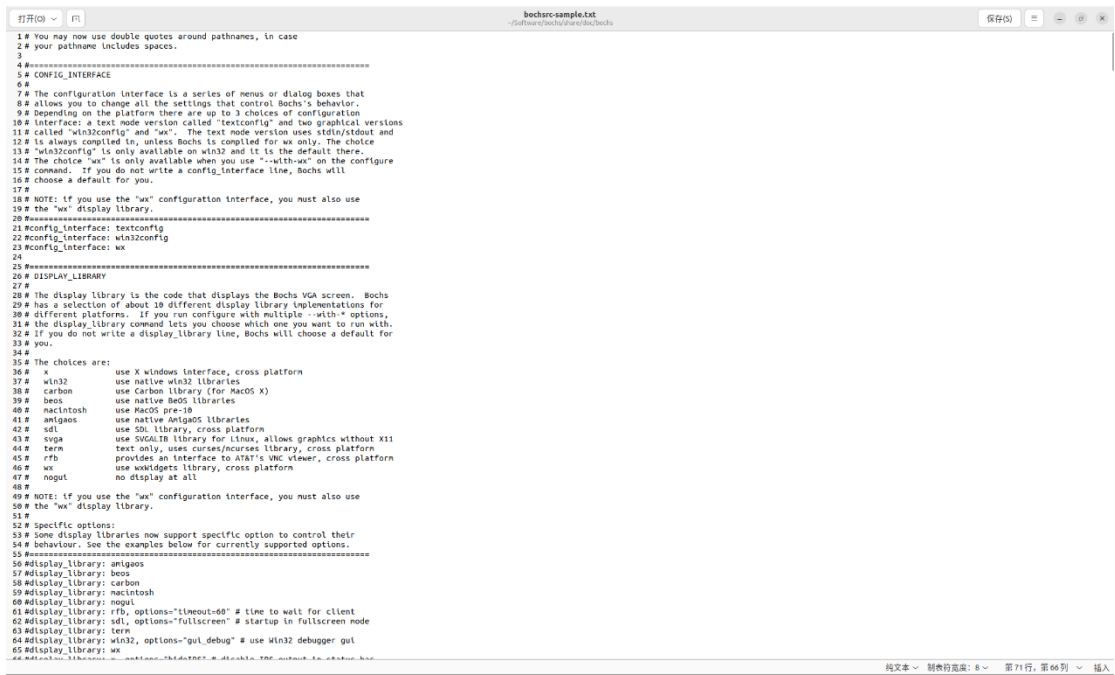
## 二、实验内容

### 2.1 掌握如何手写 Bochs 虚拟机的配置文件

- 简要介绍 Bochs 虚拟机的配置文件。

Bochs 通过读取配置文件来设置模拟环境的具体参数，这个配置文件通常以 `.bxrc` 为文件扩展名（文本文件），包含一系列的配置指令和参数，如虚拟机的 BIOS 和 VGA BIOS 路径、内存分配、启动选项、日志文件、图形界面库以及其他相关设置等。

Bochs 的安装目录 `/share/doc/bochs/` 下自带一个示例配置文件 `bochsrc-sample.txt`，便于查阅参数和改写自己的配置文件。



```
1 # You may now use double quotes around pathnames, in case
2 # your pathname includes spaces.
3
4 #####
5 CONFIG_INTERFACE
6 #
7 # The configuration interface is a series of menus or dialog boxes that
8 # allows you to change all the settings that control Bochs's behavior.
9 # Depending on the platform there are up to 3 choices of configuration
10 # interface: a text mode version called "textconfig" and two graphical versions
11 # called "win32config" and "wx". The text mode version uses stdin/stdout and
12 # is always compiled in, unless Bochs is compiled for wx only. The choice
13 # "win32config" is only available on win32 and it is the default there.
14 # The choice "wx" is only available when you use "--with-wx" on the configure
15 # command. If you do not write a config_interface line, Bochs will
16 # choose a default for you.
17 #
18 # NOTE: If you use the "wx" configuration interface, you must also use
19 # the "wx" display library.
20 #####
21 #config_interface: textconfig
22 #config_interface: win32config
23 #config_interface: wx
24
25 #####
26 DISPLAY_LIBRARY
27 #
28 # The display library is the code that displays the Bochs VGA screen. Bochs
29 # has a selection of about 19 different display library implementations for
30 # different platforms. If you run configure with multiple --with-* options,
31 # the display library command lets you choose which one you want to run with.
32 # If you do not write a display_library line, Bochs will choose a default for
33 # you.
34 #
35 # The choices are:
36 # x use X windows interface, cross platform
37 # win32 use native win32 libraries
38 # carbon use Carbon library (for MacOS X)
39 # beos use native BeOS libraries
40 # macintosh use MacOS pre-19
41 # amigaos use native AmigaOS libraries
42 # sdl use SDL library, cross platform
43 # svga use SVGA1B library for Linux, allows graphics without X11
44 # term text only, uses curses/ncurses library, cross platform
45 # rfb provides an interface to AT&T's VNC viewer, cross platform
46 # wx use wxwidgets library, cross platform
47 # noout no display at all
48 #
49 # NOTE: If you use the "wx" configuration interface, you must also use
50 # the "wx" display library.
51 #
52 # Specific options:
53 # Some display libraries now support specific option to control their
54 # behavior. See the examples below for currently supported options.
55 #####
56 #display_library: amigaos
57 #display_library: beos
58 #display_library: carbon
59 #display_library: macintosh
60 #display_library: noout
61 #display_library: rfb, options="timeout=60" # time to wait for client
62 #display_library: sdl, options="fullscreen" # startup in fullscreen mode
63 #display_library: term
64 #display_library: win32, options="gui_debug" # use Win32 debugger gui
65 #display_library: wx
66 #####
```

以下为此次实验使用的配置文件：

1. # 指定 ROM BIOS 的镜像文件路径，使用环境变量 `$BXSHARE` 表示 Bochs 的共享目录
2. romimage: file=`$BXSHARE/BIOS-bochs-latest`
- 3.
4. # 设置虚拟机的物理内存大小为 16MB
5. megs: 16
- 6.
7. # 指定 VGA BIOS 的镜像文件路径，同样使用环境变量 `$BXSHARE`
8. vgaramimage: file=`$BXSHARE/VGABIOS-lgpl-latest`
- 9.

```

10. # 配置软驱 A，设置为插入 1.44MB 的软盘镜像文件 "Image"，并且软盘状态为已插入
11. floppy: 1_44="Image", status=inserted
12.
13. # 设置模拟器的启动顺序，这里是从软驱 A 启动
14. boot: a
15.
16. # 设置日志文件的路径和文件名，模拟器的输出信息将被记录到 bochsout.txt
17. log: bochsout.txt
18.
19. # 设置图形界面库为 X 窗口系统，启用图形界面调试选项
20. display_library: x, options="gui_debug"

```

#### • 如何设置从软驱启动？

boot 定义了 Bochs 引导启动的驱动器，如软驱（floppy）、硬盘（disk）、光驱（cdrom）或网络（network），也可直接使用驱动器号“a”“c”。

修改配置文件如下所示，即为从软驱启动：

```

1. boot: floppy # 或 boot: a
2. floppy: 1_44="Image", status=inserted

```

#### • 如何设置从硬盘启动？

与从软驱启动类似，修改配置文件如下所示，即为从硬盘启动：

```

1. boot: disk
2. ata0-master: type=disk, path="disk.img", mode=flat, cylinders=20, heads=16, spt=63

```

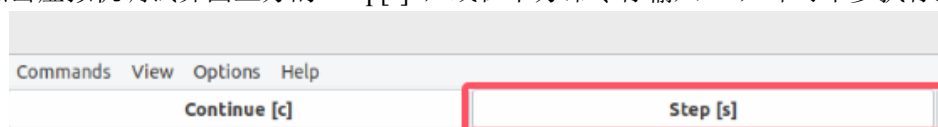
#### • 如何设置调试选项？

设置配置文件中 display\_library 参数的 options 选项为"gui\_debug"，启用图形界面调试选项。

## 2.2 掌握 Bochs 虚拟机的调试技巧

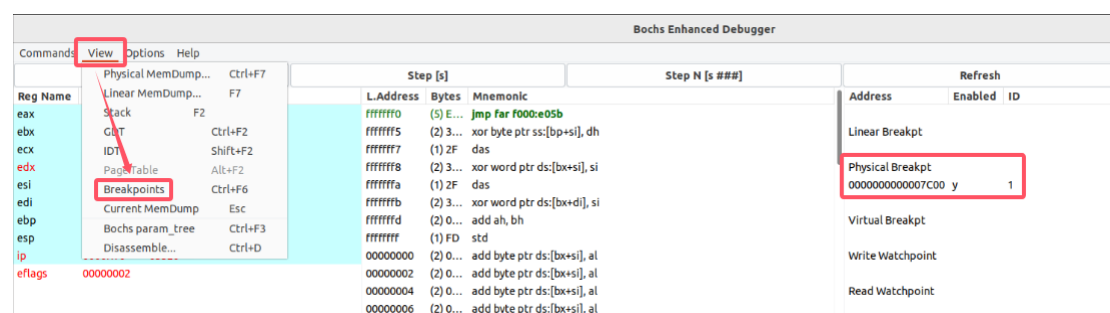
#### • 如何单步跟踪？

点击虚拟机调试界面上方的“Step[s]”，或在下方命令行输入“s”，即可单步执行指令。



#### • 如何设置断点进行调试？

在调试界面下方命令行输入“b <addr>”即可设置地址 addr 处的断点，并可通过上方 View 选项中的“Breakpoints”查看已设置的断点。



#### • 如何查看通用寄存器的值？

可在调试界面左侧查看通用寄存器的值，或在下方命令行输入“r”或“reg”查看。



- 如何查看系统寄存器的值？

选中调试界面上方 Options 选项中的“Show System Registers”，可在左侧查看系统寄存器的值，或在下方命令行使用“sreg”命令查看。



- 如何查看内存指定位置的值？

使用“x/[number][format][unit] <addr>”命令查看线性地址 addr 处的值，其中 number 为一个正整数，表示从当前地址向后显示几个地址的内容；format 表示显示的格式，例如 x（十六进制）、d（十进制）、t（二进制）、f（浮点数）、s（字符串）和 i（指令）等；unit 表示一个地址单元长度，b 表示单字节，h 表示双字节，w 表示四字节，g 表示八字节。

```
[bochs]:
0x00007c00 <bogus+ 0>: 0x00000000 0x00000000 0x00000000 0x00000000
0x00007c10 <bogus+ 16>: 0x00000000 0x00000000 0x00000000 0x00000000
0x00007c20 <bogus+ 32>: 0x00000000 0x00000000 0x00000000 0x00000000
0x00007c30 <bogus+ 48>: 0x00000000 0x00000000 0x00000000 0x00000000
0x00007c40 <bogus+ 64>: 0x00000000 0x00000000 0x00000000 0x00000000
0x00007c50 <bogus+ 80>: 0x00000000 0x00000000 0x00000000 0x00000000
0x00007c60 <bogus+ 96>: 0x00000000 0x00000000 0x00000000 0x00000000
0x00007c70 <bogus+ 112>: 0x00000000 0x00000000 0x00000000 0x00000000

x/32xw 0x7c00
```

- 如何查看各种表，如 gdt, idt, ldt 等？

点击调试界面上方 View 选项中的“GDT”“IDT”，可在右侧查看 GDT 和 IDT 表，而 GDT 中存储了 LDT 的描述符，或在下方命令行输入“info gdt”“info idt”“info ldt”查看。

Bochs Enhanced Debugger

Commands View Options Help

Physical MemDump... Ctrl+F7

Reg Name Stack F2

eax GDT

ebx IDT

ecx Shift+F2

edx Page Table Alt+F2

esi Breakpoints Ctrl+F6

edi Current MemDump Esc

ebp Bochs param\_tree Ctrl+F3

esp Disassemble... Ctrl+D

ip 00000002

eflags 00000002

gdt 00000000 (ffff)

ldt 00000000 (ffff)

Index	Base Address	Size	DPL
00	(Selector 0x0000)	0x0	0x0
01	(Selector 0x0000)	0x0	0x0
02	(Selector 0x0010)	0x0	0x0
03	(Selector 0x0010)	0x0	0x0
04	(Selector 0x0020)	0x0	0x0
05	(Selector 0x0020)	0x0	0x0
06	(Selector 0x0030)	0x0	0x0
07	(Selector 0x0030)	0x0	0x0
08	(Selector 0x0040)	0x0	0x0
09	(Selector 0x0040)	0x0	0x0
10	(Selector 0x0050)	0x0	0x0
11	(Selector 0x0050)	0x0	0x0
12	(Selector 0x0060)	0x0	0x0
13	(Selector 0x0060)	0x0	0x0
14	(Selector 0x0070)	0x0	0x0
15	(Selector 0x0070)	0x0	0x0
16	(Selector 0x0080)	0x0	0x0
17	(Selector 0x0080)	0x0	0x0
18	(Selector 0x0090)	0x0	0x0
19	(Selector 0x0090)	0x0	0x0
20	(Selector 0x00A0)	0x0	0x0
21	(Selector 0x00A0)	0x0	0x0
22	(Selector 0x00B0)	0x0	0x0
23	(Selector 0x00B0)	0x0	0x0
24	(Selector 0x00C0)	0x0	0x0
25	(Selector 0x00C0)	0x0	0x0
26	(Selector 0x00D0)	0x0	0x0
27	(Selector 0x00D0)	0x0	0x0
28	(Selector 0x00E0)	0x0	0x0
29	(Selector 0x00E0)	0x0	0x0
30	(Selector 0x00F0)	0x0	0x0
31	(Selector 0x00F0)	0x0	0x0
32	(Selector 0x0100)	0x0	0x0

GDT[0x1f4]=??? descriptor hi=0x00000000, lo=0x00000000

GDT[0x1f5]=??? descriptor hi=0x00000000, lo=0x00000000

GDT[0x1f6]=??? descriptor hi=0x00000000, lo=0x00000000

GDT[0x1f7]=??? descriptor hi=0x00000000, lo=0x00000000

GDT[0x1f8]=??? descriptor hi=0x00000000, lo=0x00000000

GDT[0x1f9]=??? descriptor hi=0x00000000, lo=0x00000000

GDT[0x1fa]=??? descriptor hi=0x00000000, lo=0x00000000

GDT[0x1fb]=??? descriptor hi=0x00000000, lo=0x00000000

GDT[0x1fc]=??? descriptor hi=0x00000000, lo=0x00000000

GDT[0x1fd]=??? descriptor hi=0x00000000, lo=0x00000000

GDT[0x1fe]=??? descriptor hi=0x00000000, lo=0x00000000

GDT[0x1ff]=??? descriptor hi=0x00000000, lo=0x00000000

You can list individual entries with 'info gdt [NUM]' or groups with 'info gdt [NUM] [NUM]'

Break CPU: Real Mode (16) t=0 IOPL=0 id vip vif ac vm rft nt of dt it lf xf al pf cf

Bochs Enhanced Debugger

Commands View Options Help

Physical MemDump... Ctrl+F7

Reg Name Stack F2

eax IDT

ebx Shift+F2

ecx Page Table Alt+F2

esi Breakpoints Ctrl+F6

edi Current MemDump Esc

ebp Bochs param\_tree Ctrl+F3

esp Disassemble... Ctrl+D

ip 00000002

eflags 00000002

gdt 00000000 (ffff)

ldt 00000000 (ffff)

Interrupt	L.Address	Base Address
00	0x0000:0x0000	0x0000:0x0000
01	0x0000:0x0000	0x0000:0x0000
02	0x0000:0x0000	0x0000:0x0000
03	0x0000:0x0000	0x0000:0x0000
04	0x0000:0x0000	0x0000:0x0000
05	0x0000:0x0000	0x0000:0x0000
06	0x0000:0x0000	0x0000:0x0000
07	0x0000:0x0000	0x0000:0x0000
08	0x0000:0x0000	0x0000:0x0000
09	0x0000:0x0000	0x0000:0x0000
0A	0x0000:0x0000	0x0000:0x0000
0B	0x0000:0x0000	0x0000:0x0000
0C	0x0000:0x0000	0x0000:0x0000
0D	0x0000:0x0000	0x0000:0x0000
0E	0x0000:0x0000	0x0000:0x0000
0F	0x0000:0x0000	0x0000:0x0000
10	0x0000:0x0000	0x0000:0x0000
11	0x0000:0x0000	0x0000:0x0000
12	0x0000:0x0000	0x0000:0x0000
13	0x0000:0x0000	0x0000:0x0000
14	0x0000:0x0000	0x0000:0x0000
15	0x0000:0x0000	0x0000:0x0000
16	0x0000:0x0000	0x0000:0x0000
17	0x0000:0x0000	0x0000:0x0000
18	0x0000:0x0000	0x0000:0x0000
19	0x0000:0x0000	0x0000:0x0000
1A	0x0000:0x0000	0x0000:0x0000
1B	0x0000:0x0000	0x0000:0x0000
1C	0x0000:0x0000	0x0000:0x0000
1D	0x0000:0x0000	0x0000:0x0000
1E	0x0000:0x0000	0x0000:0x0000
1F	0x0000:0x0000	0x0000:0x0000
20	0x0000:0x0000	0x0000:0x0000

IDT[0x1f4]=??? descriptor hi=0x00000000, lo=0x00000000

IDT[0x1f5]=??? descriptor hi=0x00000000, lo=0x00000000

IDT[0x1f6]=??? descriptor hi=0x00000000, lo=0x00000000

IDT[0x1f7]=??? descriptor hi=0x00000000, lo=0x00000000

IDT[0x1f8]=??? descriptor hi=0x00000000, lo=0x00000000

IDT[0x1f9]=??? descriptor hi=0x00000000, lo=0x00000000

IDT[0x1fa]=??? descriptor hi=0x00000000, lo=0x00000000

IDT[0x1fb]=??? descriptor hi=0x00000000, lo=0x00000000

IDT[0x1fc]=??? descriptor hi=0x00000000, lo=0x00000000

IDT[0x1fd]=??? descriptor hi=0x00000000, lo=0x00000000

IDT[0x1fe]=??? descriptor hi=0x00000000, lo=0x00000000

IDT[0x1ff]=??? descriptor hi=0x00000000, lo=0x00000000

You can list individual entries with 'info idt [NUM]' or groups with 'info idt [NUM] [NUM]'

Break CPU: Real Mode (16) t=0 IOPL=0 id vip vif ac vm rft nt of dt it lf xf al pf cf

```

LDT[0x1ff4]=??? descriptor hi=0x00000000, lo=0x00000000
LDT[0x1ff5]=??? descriptor hi=0x00000000, lo=0x00000000
LDT[0x1ff6]=??? descriptor hi=0x00000000, lo=0x00000000
LDT[0x1ff7]=??? descriptor hi=0x00000000, lo=0x00000000
LDT[0x1ff8]=??? descriptor hi=0x00000000, lo=0x00000000
LDT[0x1ff9]=??? descriptor hi=0x00000000, lo=0x00000000
LDT[0x1ffa]=??? descriptor hi=0x00000000, lo=0x00000000
LDT[0x1ffb]=??? descriptor hi=0x00000000, lo=0x00000000
LDT[0x1ffc]=??? descriptor hi=0x00000000, lo=0x00000000
LDT[0x1ffd]=??? descriptor hi=0x00000000, lo=0x00000000
LDT[0x1ffe]=??? descriptor hi=0x00000000, lo=0x00000000
LDT[0x1fff]=??? descriptor hi=0x00000000, lo=0x00000000
You can list individual entries with 'info ldt [NUM]' or groups with 'info ldt [NUM] [NUM]'

```

- 如何查看 TSS?

在调试界面下方命令行输入“info tss”即可查看，也可以通过 GDT 表查看。

```

tr:s=0x0, base=0x00000000, valid=1
ss:esp(0): 0x0000:0x00000000
ss:esp(1): 0x0000:0x00000000
ss:esp(2): 0x0000:0x00000000
cr3: 0x00000000
eip: 0x00000000
eflags: 0x00000000
cs: 0x0000 ds: 0x0000 ss: 0x0000
es: 0x0000 fs: 0x0000 gs: 0x0000
eax: 0x00000000 ebx: 0x00000000 ecx: 0x00000000 edx: 0x00000000
esi: 0x00000000 edi: 0x00000000 ebp: 0x00000000 esp: 0x00000000
ldt: 0x0000
i/o map: 0x0000
info tss

```

- 如何查看栈中的内容?

点击调试界面上方 View 选项中的“STACK”，可在右侧查看栈中内容，或在下方命令行使用“print-stack”命令查看

The screenshot shows the Bochs Enhanced Debugger interface. In the 'View' menu, 'STACK' is selected. The main window displays a table of memory addresses and their contents, organized into columns: L.Address, Bytes, Mnemonic, Step N [s ##], L.Address, Value, and Refresh. The stack content is shown as a series of memory addresses (e.g., 00000000, 00000002, 00000004, etc.) and their corresponding values (e.g., 0, 0, 0, etc.).

Reg Name	Value
eax	00000000
ebx	00000000
ecx	00000000
edx	00000000
esi	00000000
edi	00000000
ebp	00000000
esp	00000000
ip	00000000
eflags	00000000
gdt	00000000
idt	00000000

- 如何在内存指定地方进行反汇编?

使用“u [/num] [start] [end]”命令反汇编地址 start 到 end 之间的代码，若不指定地址参数则反汇编当前 eip 寄存器指向的代码，num 用于指定处理的代码量。



```

00007c06: (      ); add byte ptr ds:[bx+si], al; 0000
00007c08: (      ); add byte ptr ds:[bx+si], al; 0000
00007c0a: (      ); add byte ptr ds:[bx+si], al; 0000
00007c0c: (      ); add byte ptr ds:[bx+si], al; 0000
00007c0e: (      ); add byte ptr ds:[bx+si], al; 0000
00007c10: (      ); add byte ptr ds:[bx+si], al; 0000
00007c12: (      ); add byte ptr ds:[bx+si], al; 0000
00007c14: (      ); add byte ptr ds:[bx+si], al; 0000
00007c16: (      ); add byte ptr ds:[bx+si], al; 0000
00007c18: (      ); add byte ptr ds:[bx+si], al; 0000
00007c1a: (      ); add byte ptr ds:[bx+si], al; 0000
00007c1c: (      ); add byte ptr ds:[bx+si], al; 0000
00007c1e: (      ); add byte ptr ds:[bx+si], al; 0000

```

u 0x7c00 0x7c20

## 2.3 计算机引导程序

- 如何查看 0x7c00 处被装载了什么？

从 0x7c00 处开始一般为引导扇区的内容，可以使用“x/[number][format][unit] 0x7c00”命令，如“x/32xw 0x7c00”以十六进制格式、四字节地址单元长度查看其后 32 个地址内容。

- 如何把真正的内核程序从硬盘或软驱装载到自己想要放的地方？

利用引导加载程序 boot 可将内核程序从存储设备加载到内存中，并初始化系统进入保护模式。通过阅读 boot.s 汇编文件可知，boot 程序首先设置段寄存器和堆栈，然后通过 BIOS 中断 int 0x13 从软盘加载操作系统内核到内存地址 0x10000 处，其中 mov dx,#0x0000 设置软盘起始扇区，mov cx,#0x0002 定义读取扇区数，mov ax,#SYSSEG 和 mov es,ax 指定内存加载地址。成功后使用 cli 指令禁用中断（以防切换到保护模式时发生中断），接着通过 lgdt gdt\_48 和 lidt idt\_48 加载 GDT 和 IDT，然后设置 cr0 寄存器以启用保护模式，并最终通过 jmp 0,8 跳转到内核入口点，完成启动过程。若加载失败，则进入 die 标签的无限循环。

- 如何查看实模式的中断程序？

查看中断向量表。中断向量表是 BIOS 和操作系统用来处理中断请求的跳转地址列表，在实模式下位于内存的前 1024 字节，每个中断向量占用 4 个字节（前 2 个字节为偏移地址，后 2 个字节为段地址），对应一个中断处理程序。在 Bochs 虚拟机中可以使用“info ivt”或“info idt”命令查看并找到目标中断处理程序的地址，之后使用“x/[number][format][unit] <addr>”命令查看具体内容。

- 如何静态创建 gdt 与 idt？

首先定义 GDT 和 IDT 的描述符结构，每个描述符通常由 8 个字节组成，前 6 个字节为描述符本身，后 2 个字节为访问控制字（AC）和限制字段；接着创建 GDT 和 IDT 的表，其包含指向描述符的指针；然后使用 lgdt 指令加载 GDT，使用 lidt 指令加载 IDT；最后启用保护模式。

- 如何从实模式切换到保护模式？

首先设置并使用 lgdt 指令加载 GDT，其至少包含两个描述符，一个对应代码段，一个对应数据段，用于保护模式下的内存访问；接着使用 cli 指令禁用中断，以防止在切换过程中被中断；然后修改 cr0 寄存器，将 cr0 寄存器的第 0 位（PE 位）设置为 1，以启用保护模式；最后执行一个远跳转指令，这将迫使 CPU 刷新其内部缓存的段寄存器，从而加载 GDT 中的新描述符。这样就成功切换到了保护模式，可以开始编写保护模式下的代码，注意需要重新设置所有段寄存器。

- 调试跟踪 jmp 0,8，解释如何寻址。

jmp 0,8 位于 boot.s 文件中，用于跳转至内核程序入口点，将控制权交给操作系统内核，完成从实模式到保护模式的转换。

打开 Bochs 的调试界面，在 0x7c00 处设置断点，点击上方 Continue[c] 开始运行，程序运行至断点后可从中间查看 0x7c00 及之后的汇编代码（即 boot.s 汇编代码）。查找 jmp 0,8 发现位于地址 0x7c4c 处，则在此地址再设置一个断点，继续运行程序，直至运行至此处。使用 Step[s] 进行单步跟踪，发现跳转至地址 0x0000 处。

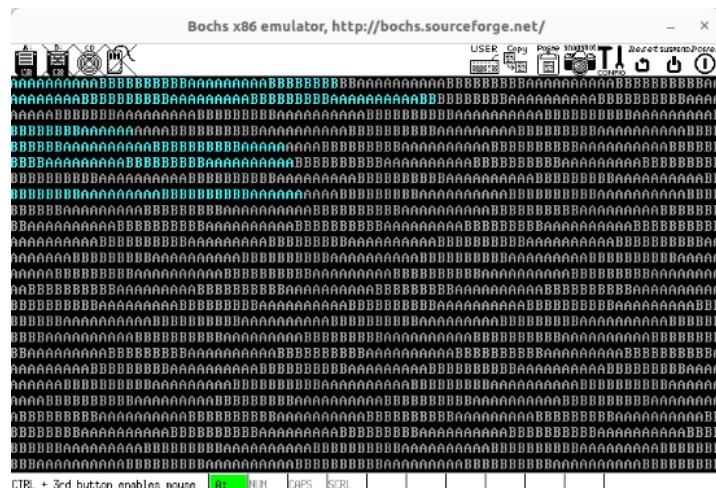
jmp 是一个用于 x86 实模式下的段间跳转指令，采用段基址+偏移量的寻址方式，其指令格式为“jmp <offset>”，其中 offset 是一个 16 位的偏移量，在执行 jmp 指令时，CPU 会将当前的代码段基址与这个偏移量相加，得到跳转的目标地址。

对于“jmp 0,8”，偏移量为 0，后面的 8 为段选择子，被加载到 cs 寄存器中，而 8 的二进制表示为 1000，则 RPL=0，TI=0（表示 GDT），Index=1（表示索引为 1，即 GDT 第二个表项内核代码段），而 GDT 内核代码段的基址为 0，故实际上跳转至 base(0)+offset(0)=0 处，即为 head.s 文件存放在内存中的地址，而系统当前还未开启分页机制（可以查看 cr0 寄存器第 0 位和第 31 位），因此这个 0 就是真实的物理地址。

## 三、实验过程

### 3.1 head.s 的工作原理

在 Linux 0.00 中，head.s 文件的主要作用是负责初始化和引导处理器到内核的核心部分，设置硬件环境、处理器状态和内存布局，为内核的 C 代码执行做好准备。其包含两个任务——任务 0 和任务 1，运行时会先执行任务 0 打印 10 个 A，然后切换到任务 B 打印 10 个 B，如此循环往复。



head.s 工作原理包括以下几方面：

（1）设置段寄存器和堆栈（13-16 行）

- movl \$0x10, %eax 和 mov %ax, %ds 设置数据段寄存器 DS 为 0x10（内核数据段选择子）；
- lss init\_stack,%esp 加载堆栈段选择子和堆栈指针，初始化堆栈。

（2）设置 GDT 和 IDT

• 19-20 行 call setup\_idt 和 call setup\_gdt 调用函数设置中断描述符表和全局描述符表，函数中 78 行和 94 行 lgdt lgdt\_opcode 和 lidt lidt\_opcode 指令加载 GDT 和 IDT 的指针。

（3）设置时钟中断

- 29-36 行通过端口操作设置 8253 时钟芯片，使其每隔 10 毫秒产生一个中断。

（4）设置中断描述符

• 39-51 行 movl \$0x00080000 ~ movl %edx,4(%esi) 设置时钟中断和系统调用中断的描述符，这些中断用于操作系统的调度和系统调用。

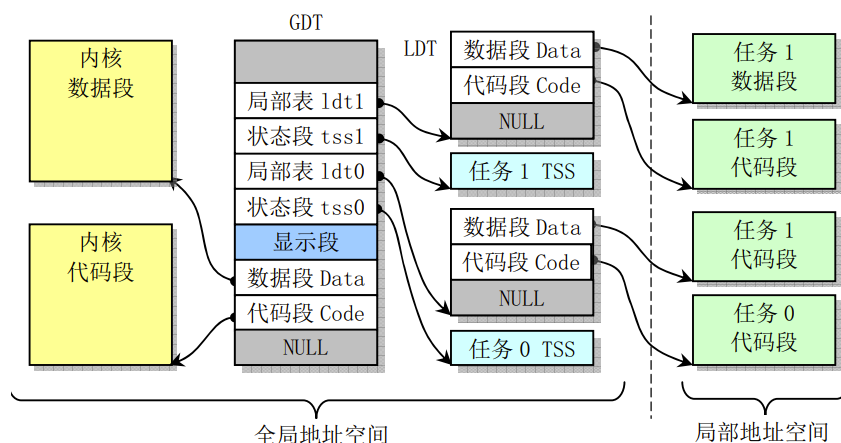
（5）切换到用户模式（60-74 行）

- pushfl 和 popfl 用于清除 eflags 寄存器中的 IF 标志，防止中断；
- movl \$TSS0\_SEL, %eax 和 ltr %ax 设置任务状态段寄存器 TSS；
- movl \$LDT0\_SEL, %eax 和 lldt %ax 设置本地描述符表寄存器 LDT；
- sti 启用中断；

- 通过 `iret` 指令从中断返回，切换至用户模式。
- (6) 中断处理函数
- 121-130 行 `ignore_int` 是默认的中断处理函数，用于处理未定义的中断；
  - 134-151 行 `timer_interrupt` 是时钟中断处理函数，用于在每个时钟中断时切换任务；
  - 155-169 行 `system_interrupt` 是系统调用中断处理函数。
- (7) 任务切换
- 设置 `current` 变量用于跟踪当前任务。
  - 238-254 行实现 `task0` 和 `task1` 两个任务，分别打印字符'A'和'B'。
- (8) 全局描述符表 (GDT) 和中断描述符表 (IDT) (176-196 行)
- 184-194 行 `gdt` 和 `idt` 定义了 GDT 和 IDT 的条目，包括空描述符、代码段、数据段、任务状态段 (TSS) 和本地描述符表 (LDT)。
- (9) 任务状态段 (TSS) 和本地描述符表 (LDT)
- 207-213 行 `tss0` 和 225-232 行 `tss1` 定义了两个任务状态段，包括堆栈指针、寄存器状态等；
  - 203-205 行 `ldt0` 和 221-223 行 `ldt1` 定义了两个任务的本地描述符表。

### 3.2 head.s 的内存分布状况

head.s 的内存地址空间如下图所示：



head.s 各数据段、代码段、堆栈段的起始与终止的内存地址如下表所示：

段名称	起始地址	终止地址
内核数据段	0x000000	0x7FFFFFFF
内核代码段	0x000000	0x7FFFFFFF
堆栈段	0x000000	0x7FFFFFFF
显示段	0x0B8000	0x0BAFFF
任务 0 数据段	0x000000	0x3FFFFFFF
任务 0 代码段	0x000000	0x3FFFFFFF
任务 1 数据段	0x000000	0x3FFFFFFF
任务 1 代码段	0x000000	0x3FFFFFFF

- 内核数据段与内核代码段
- 运行程序，查看 GDT 表，内核代码段与内核数据段位于 GDT 中第二、三个表项，其段选择子分别为 `0x8` 和 `0x10`。由下图可以看到内核代码段与内核数据段基址均为 `0x0`，大小为 `0x7FFFFFFF`，即终止地址为 `0x7FFFFFFF`。





这几行代码位于任务切换函数中，57-62 行将局部空间数据段（堆栈段）选择子、堆栈指针（栈底）、标志寄存器的值、局部空间代码段选择子和代码指针压入栈，62 行 `iret` 指令执行后从中断处理程序返回，会从栈中弹出这五个值，并分别赋给 `ss`、`esp`、`eflags`、`cs` 和 `eip` 寄存器，用于恢复之前的程序状态。

### 3.4 `iret` 执行后 `pc` 如何找到下一条指令

`eip` 寄存器用于存储当前正在执行的指令的内存地址，当 CPU 执行完一条指令后，`eip` 会自动递增，指向下一条待执行的指令。`cs` 寄存器用于存储当前代码段选择子，可通过查 GDT 表找到段基址。

`iret` 指令用于从中断处理程序返回，并恢复到中断发生前的程序状态，其执行后会从堆栈中弹出以下值：

- `cs` 和 `eip` 寄存器的值，以恢复被中断程序的代码指针。
- `eflags` 寄存器的值，以恢复被中断程序的标志位。
- `ss` 和 `esp` 寄存器的值，以恢复被中断程序的堆栈指针。

`iret` 指令会 `cs` 和 `eip` 寄存器的值组合成一个 32 位的线性地址来恢复程序计数器 PC，从而找到下一条待执行的指令。

### 3.5 `iret` 执行前后栈的变化

在 0x9d 处设置一个断点，单步执行至 `iret` 指令执行之前，查看 Stack 的内容如下，此时栈顶为先前指令压入栈的值：

Bochs Enhanced Debugger									
Continue [c]			Step [s]			Step N [s ###]			Refresh
Reg Name	Hex Value	Decimal	L.Address	Bytes	Mnemonic	L.Address	Value	(dec.)	
eax	00000028	40	0000009d	(2) 6...	push 0x00000017	000000c4	000010e0	4320	
ebx	00000000	0	0000009f	(5) 6...	push 0x00000bd8	000000c8	0000000f	15	
ecx	00000080	128	000000a4	(1) 9C	pushfd	000000cc	00000246	582	
edx	0000ef00	61184	000000a5	(2) 6...	push 0x0000000f	000000d0	00000bd8	3032	
esi	00000598	1432	000000a7	(5) 6...	push 0x000010e0	000000d4	00000017	23	
edi	00000998	2456	000000ac	(1) CF	iretd	000000d8	00000bd8	3032	
ebp	00000000	0	000000ad	(7) 0...	lgdt ds:0x18c	000000dc	90660010	-1872363504	
esp	00000bc4	3012	000000b4	(1) C3	ret	000000e0	00000000	0	
eip	000000ac	172	000000b5	(6) 8...	lea edx, dword ptr ds:0x114	000000e4	00000000	0	
eflags	00000246		000000b6	(5) B...	mov eax, dx	000000e8	000003ff	1023	
cs	0008		000000c0	(3) 6...	mov ax, dx	000000ec	00cf0a00	12646912	
ds	0010		000000c3	(4) 6...	mov dx, 0x8e00	000000f0	000003ff	1023	
es	0010		000000c7	(6) 8...	lea edi, dword ptr ds:0x198	000000f4	00cf0200	12644864	
ss	0010		000000cd	(5) B...	mov ecx, 0x00000100	000000f8	00000000	0	

单步执行程序，在 `iret` 指令执行后再次查看 Stack 内容如下：

Bochs Enhanced Debugger									
Continue [c]			Step [s]			Step N [s ###]			Refresh
Reg Name	Hex Value	Decimal	L.Address	Bytes	Mnemonic	L.Address	Value	(dec.)	
eax	00000028	40	000010e0	(5) B...	mov eax, 0x00000017	000000d8	00000bd8	3032	
ebx	00000000	0	000010e5	(2) 8...	mov ds, ax	000000dc	90660010	-1872363504	
ecx	00000080	128	000010e7	(2) B...	mov al, 0x41	000000e0	00000000	0	
edx	0000ef00	61184	000010e9	(2) C...	int 0x80	000000e4	00000000	0	
esi	00000598	1432	000010eb	(5) B...	mov ecx, 0x000000ff	000000e8	000003ff	1023	
edi	00000998	2456	000010fd	(2) E...	loop -2 (0x000010fd)	000000ec	00cf0b00	12647168	
ebp	00000000	0	000010fe	(2) E...	jmp -2 (0x000010fe)	000000f0	000003ff	1023	
esp	00000bd8	3032	000010ff	(5) B...	mov eax, 0x00000017	000000f4	00cf0300	12645120	
eip	000010e0	4320	000010f9	(2) B...	mov ds, ax	000000f8	00000000	0	
eflags	00000246		000010fb	(2) B...	mov al, 0x42	000000fc	00000e60	3680	
cs	000f		000010fd	(2) C...	int 0x80	00000000	00000010	16	
ds	0000		000010ff	(5) B...	mov ecx, 0x000000ff	00000004	00000000	0	
es	0000		00001104	(2) E...	loop -2 (0x00001104)	00000008	00000000	0	
ss	0017		00001106	(2) E...	jmp -20 (0x00001104)	0000000c	00000000	0	
fs	0000		00001108	(2) 0...	add byte ptr ds:[eax], al	00000010	00000000	0	
gs	0000		0000110a	(2) 0...	add byte ptr ds:[eax], al	00000014	00000000	0	

可以看到栈顶向下移动，弹出了 5 个值，即 `iret` 指令之前压入栈的 5 个值，其余内容没变。结合左侧寄存器的值，发现 `cs`、`eip`、`ss` 和 `esp` 寄存器发生改变，且与弹出的值相对应。

### 3.6 当任务进行系统调用时（即 `int 0x80` 时）栈的变化情况

`int $0x80` 是一条 AT&T 语法的中断指令，用于 Linux 的系统调用，而 `head.s` 中任务 0 代码段会执行这个系统调用。在上面的 `iret` 指令处（0xac）设置一个断点，其执行后程序转到任务 0 的第一条指令 0x10e0 处，然后单步执行至 `int $0x80` 指令之前，此时任务 0 已将系统调用号和相关参数加载到寄存器中。

查看 Stack 内容如下，包含了任务 0 正常执行的栈帧，包括函数调用参数、局部变量等：

Bochs Enhanced Debugger									
Continue [c]			Step [s]			Step N [s ###]			Refresh
Reg Name	Hex Value	Decimal	L.Address	Bytes	Mnemonic				
eax	00000041	65	000010e0	(5) B...	mov eax, 0x00000017				
ebx	00000000	0	000010e5	(2) B...	mov ds, ax				
ecx	00000000	0	000010e7	(2) B...	mov al, 0x41				
edx	0000e0f0	61184	000010e9	(2) C...	int 0x80				
esi	00000598	1432	000010eb	(5) B...	mov ecx, 0x000000ff				
edi	00000998	2456	000010f0	(2) E...	loop -2 (0x000010f0)				
ebp	00000000	0	000010f2	(2) E...	jmp -20 (0x000010e0)				
esp	00000bd8	3032	000010f4	(5) B...	mov eax, 0x00000017				
eip	000010e9	4329	000010f9	(2) B...	mov ds, ax				
eflags	00000246		000010fb	(2) B...	mov al, 0x42				
cs	000f		000010fd	(2) C...	int 0x80				
ds	0017		000010ff	(5) B...	mov ecx, 0x000000ff				
es	0000		00001104	(2) E...	loop -2 (0x00001104)				
ss	0017		00001106	(2) E...	jmp -20 (0x000010f4)				
fs	0000		00001108	(2) 0...	add byte ptr ds:[eax], al				
gs	0000		0000110a	(2) 0...	add byte ptr ds:[eax], al				
gdt	00000998 ( 3f)		0000110c	(2) 0...	add byte ptr ds:[eax], al				
ldtr	00000198 ( 7ff)		0000110e	(2) 0...	add byte ptr ds:[eax], al				
ldtr	0be0		00001110	(2) 0...	add byte ptr ds:[eax], al				
tr	0bf8		00001112	(2) 0...	add byte ptr ds:[eax], al				
cr0	60000011		00001114	(2) 0...	add byte ptr ds:[eax], al				
cr2	00000000		00001116	(2) 0...	add byte ptr ds:[eax], al				
cr3	00000000		00001118	(2) 0...	add byte ptr ds:[eax], al				
cr4	00000000		0000111a	(2) 0...	add byte ptr ds:[eax], al				
efer	00000000		0000111c	(2) 0...	add byte ptr ds:[eax], al				
			0000111e	(2) 0...	add byte ptr ds:[eax], al				
			00001120	(2) 0...	add byte ptr ds:[eax], al				
			00001122	(2) 0...	add byte ptr ds:[eax], al				
			00001124	(2) 0...	add byte ptr ds:[eax], al				
			00001126	(2) 0...	add byte ptr ds:[eax], al				
			00001128	(2) 0...	add byte ptr ds:[eax], al				
			0000112a	(2) 0...	add byte ptr ds:[eax], al				
			0000112c	(2) 0...	add byte ptr ds:[eax], al				

(0) Breakpoint 1, 0x000000ac in ?? ()

单步执行程序，在 int \$0x80 指令执行后再次查看 Stack 内容如下：

Bochs Enhanced Debugger									
Continue [c]			Step [s]			Step N [s ###]			Refresh
Reg Name	Hex Value	Decimal	L.Address	Bytes	Mnemonic				
eax	00000041	65	00000166	(1) 1E	push ds				
ebx	00000000	0	00000167	(1) 52	push edx				
ecx	00000000	0	00000168	(1) 51	push ecx				
edx	0000e0f0	61184	00000169	(1) 53	push ebx				
esi	00000598	1432	0000016a	(1) 50	push eax				
edi	00000998	2456	0000016b	(5) B...	mov edx, 0x00000010				
ebp	00000000	0	00000170	(2) B...	mov ds, dx				
esp	00000e4c	3660	00000172	(5) E...	call -146 (0x000000e5)				
eip	00000166	358	00000177	(1) 58	pop eax				
eflags	00000246		00000178	(1) 58	pop ebx				
cs	000f		00000179	(1) 59	pop ecx				
ds	0017		0000017a	(1) 5A	pop edx				
es	0000		0000017b	(1) 1F	pop ds				
ss	0010		0000017c	(1) CF	iretd				
fs	0000		0000017d	(2) 0...	add byte ptr ds:[eax], al				
gs	0000		0000017f	(2) 0...	add byte ptr ds:[eax], al				
gdt	00000998 ( 3f)		00000181	(2) 0...	add dword ptr ds:[eax], eax				
ldtr	00000198 ( 7ff)		00000183	(2) 0...	add byte ptr ds:[eax], al				
ldtr	0be0		00000185	(1) 90	nop				
tr	0bf8		00000186	(2) F...	inc dword ptr ds:[edi]				
cr0	60000011		00000188	(1) 98	cwde				
cr2	00000000		00000189	(2) 0...	add dword ptr ds:[eax], eax				
cr3	00000000		0000018b	(2) 0...	add byte ptr ds:[edi], bh				
cr4	00000000		0000018d	(6) 0...	add byte ptr ds:[eax-192937983], bl				
efer	00000000		00000193	(2) B...	mov dh, 0x00				
			00000195	(2) 0...	add byte ptr ds:[eax], al				
			00000197	(3) 0...	add byte ptr ds:[ecx+eax], dl				
			0000019a	(2) 0...	or byte ptr ds:[eax], al				
			0000019c	(6) 0...	add byte ptr ds:[esi+18087936], cl				
			000001a2	(2) 0...	or byte ptr ds:[eax], al				
			000001a4	(6) 0...	add byte ptr ds:[esi+18087936], cl				
			000001aa	(2) 0...	or byte ptr ds:[eax], al				
			000001ac	(6) 0...	add byte ptr ds:[esi+18087936], cl				

(0) Breakpoint 1, 0x000000ac in ?? ()

可以看到 SS:ESP 切换到了 task0 内核栈 0x10:0x0e4c 处，并且 CS:EIP 跳转到了系统调用中断处理程序入口地址 0x08:0x0166 处。并且原来的用户栈顶地址 0x17:0x0bd8 和中断返回地址 0x0f:0x10eb 被压到了内核栈顶。

当任务 0 执行 int \$0x80 指令时，CPU 会将标志寄存器的值、代码段选择子、返回地址系统调用号和参数压入栈，进入内核态后，内核会根据系统调用号，从栈上获取参数，执行相应的系统调用服务。系统调用处理程序执行完后，通常会返回将返回值存储在 eax 寄存器中，之后使用 iret 指令返回到用户态，将栈上的值弹出，恢复到 int \$0x80 指令执行前的状态。

## 四、实验结果

通过修改配置文件，配置 Bochs 虚拟机的环境，并掌握了 Bochs 的简单调试方法和技巧，对 Linux 0.00 的引导加载程序和内核程序进行调试。对 GDT 和 IDT、各寄存器作用、各数据段和代码段的内存空间和段选择子、中断处理程序、任务切换和模式切换有了更深刻的理解，了解了操作系统启动的步骤和过程。