调试分析 Linux0.00 引导程序

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

实验内容

掌握如何手写 Bochs 虚拟机的配置文件

- 简介 Bochs 虚拟机的配置文件

（1）该文件指定了`BIOS`的位置，它位于`0xf0000`到`0xfffff`之间的内存空间中。可以使用`$BXSHARE/BIOS-bochs-latest`作为`BIOS`文件的路径。

```
#=======================================================================
romimage: file=$BXSHARE/BIOS-bochs-latest
#, address=0xf0000
#romimage: file=$BXSHARE/BIOS-bochs-2-processors, address=0xf0000
#=======================================================================
```

（2）该文件指定了分配给虚拟机的内存大小，默认值是 32MB，这里设置为 16MB。

```
#=======================================================================
#megs: 64
megs: 16
```

（3）该文件指定了`VGA ROM BIOS`的位置，它位于内存空间中的`C0000`地址。

```
#=======================================================================
vgaromimage: file=$BXSHARE/VGABIOS-lgpl-latest
#vgaromimage: $BXSHARE\VGABIOS-elpin-2.40
```

（4）该文件指定了软盘映像文件的位置，可以使用`"Image"`作为软盘映像文件的路径。

```
#=======================================================================
#floppya: 1_44=/dev/fd0, status=inserted
#floppya: 1_44=a:, status=inserted            # for win32
floppya: 1_44="Image", status=inserted
```

（5）该文件指定了启动驱动器的位置，这里指定从软盘启动。

```
#=======================================================================
boot: a
#boot: c
```

（6）该文件指定了调试信息的输出位置，可以将其输出到`bochsout.txt`文件中。

```
#=======================================================================
#log: /dev/null
log: bochsout.txt
```

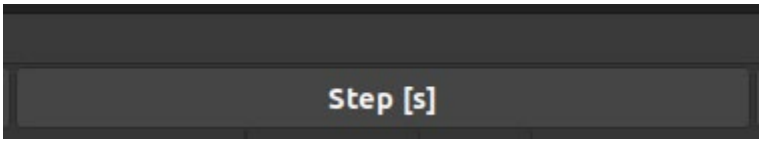
（7）可视化配置，因为本人在 ubuntu 上做实验，因此选择 Linux 的配置

```
# windows
# config_interface: win32config
# display_library: win32, options="gui_debug"

# Linux
display_library: x, options="gui_debug"

# MacOS
# display_library: sdl2
```

- 如何设置从软驱启动
  上述设置就是从软驱启动：
  floppya: 1_44="Image", status=inserted； boot： a
- 如何设置从硬盘启动
  打开`bochsrc`文件，查找`ata0-master`部分的配置。这里是用于模拟硬盘的设置。确认`ata0-master`的`type`配置为`disk`，表示使用硬盘模式。
  设置`ata0-master`的`path`配置为想要从中启动的硬盘映像文件的路径。
  其中，`path`配置项需要根据实际情况修改，`mode`通常设为`flat`，表示使用扁平映像模式。`cylinders`、`heads`和`spt`则是设置磁盘的几何参数，通常不需要修改。
  最终设置完毕如下所示：
  `ata0-master: type=disk, path="$OSLAB_PATH/hdc-0.11.img", mode=flat, cylinders=204, heads=16, spt=38`
  保存`bochsrc`文件并重新启动`Bochs`，此时`Bochs`应该会从指定的硬盘启动。
- 如何设置调试选项
  在 log 中指定一个文件路径存放日志信息
  在可视化配置中进行相应的配置，如上述配置是基于 Linux 的配置，若在 windows 和 MacOS 系统运行，可取消相应的注释

掌握 Bochs 虚拟机的调试技巧

- 如何单步跟踪？



- 如何设置断点进行调试？

Eg： b 0x7c00

Physical Breakpt		
0000000000007C00	y	1

再点击 continue：

Continue [c] | Step [s]

Reg Name	Hex Value	Decimal	L.Address	Bytes	Mnemonic
eax	0000aa55	43605	00007c00	(5) E...	jmpf 0x07c0:0005
ebx	00000000	0	00007c05	(2) 8...	mov ax, cs

- 如何查看通用寄存器的值？

Reg Name	Hex Value	Decimal
eax	0000aa55	43605
ebx	00000000	0
ecx	00090000	589824
edx	00000000	0
esi	000e0000	917504
edi	0000ffac	65452
ebp	00000000	0
esp	0000ffd6	65494
ip	00007c00	31744

- 如何查看系统寄存器的值？

eflags	00000082
cs	0000
ds	0000
es	0000
ss	0000
fs	0000
gs	0000
gdtr	000f9ad7 (30)
idtr	00000000 (3ff)
cr0	60000010
cr2	00000000
cr3	00000000
cr4	00000000
efer	00000000

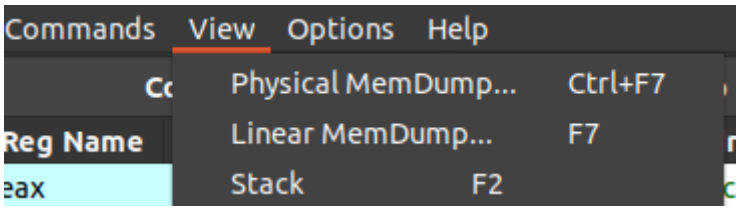
- 如何查看内存指定位置的值？

  输入`x /nuf <addr>`命令，其中`<addr>`是要查看的内存地址，按`Enter`键后将显示该地址处的内存值。

- 如何查看各种表，如 gdt， idt， ldt 等？

`info gdt`：显示全局描述符表（GDT）的内容。
`info idt`：显示中断描述符表（IDT）的内容。
`info ldt`：显示本地描述符表（LDT）的内容。

- 如何查看 TSS?
  Info tss
- 如何查看栈中的内容?

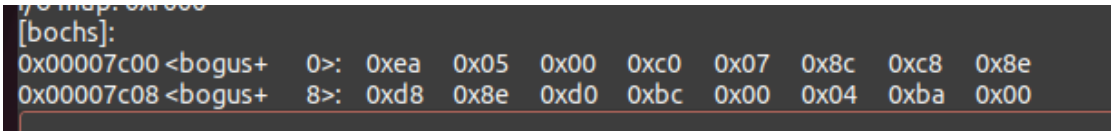


- 如何在内存指定地方进行反汇编?
  u [addr]

计算机引导程序

1. 如何查看 0x7c00 处被装载了什么?

设置断点，程序执行到 0x7c00 后:
x/16bx 0x7c00



2. 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;

要将真正的内核程序从硬盘或软驱加载到指定位置，需要编写引导加载程序（Bootloader）或操作系统的启动代码来执行这个过程。以下是一般的步骤:

引导加载程序的编写:

编写引导加载程序的代码，该程序位于引导扇区（通常是硬盘的第一个扇区，也可以是软驱的引导扇区）。引导加载程序的代码应负责初始化硬件、加载内核程序到内存，然后跳转到内核的入口点开始执行。

选择加载位置:

决定将内核程序加载到内存中的哪个位置。通常内核会被加载到一个特定的内存地址，该地址在编写引导加载程序时被指定。

读取磁盘内容:

引导加载程序需要使用适当的读取磁盘的函数或系统调用来从硬盘或软驱读取内核程序的内容。这通常涉及到磁盘驱动程序或相关的系统库函数。

内存分配和加载:

在内存中分配足够的空间来容纳内核程序，并将从磁盘读取的内容加载到分配的内存空间中。可以使用适当的内存管理函数或系统调用来完成这一任务。

执行内核:

在加载完内核程序后，引导加载程序需要跳转到内核的入口点，开始执行内核代码。

3. 如何查看实模式的中断程序?

获取中断向量:

在实模式下，中断向量表位于内存地址`0x0000-0x03FF`。每个中断向量占用 4 个字节，其中前两个字节是段地址，后两个字节是偏移地址。要查看特定中断的程序，首先需要确定该中断的中断号，然后计算中断向量在内存中的地址。

计算中断向量的物理地址:

使用中断号乘以 4 来计算中断向量在中断向量表中的偏移量。将偏移量与中断向量表的基地址`0x0000`相加，得到中断向量的物理地址。

查看中断程序:

使用调试工具或反汇编工具打开内存地址对应的中断程序的物理地址。对中断程序进行反汇编，以查看其汇编指令和执行逻辑。

4. 如何静态创建 gdt 与 idt？

定义`gdt`和`idt`的结构

创建`gdt`和`idt`的实例

初始化`gdt`和`idt`的描述符

将`gdt`和`idt`加载到处理器

5. 如何从实模式切换到保护模式?

关中断->加载 gdt->设置控制寄存器->切换到保护模式

6. 调试跟踪 jmpi 0,8 ， 解释如何寻址?

首先，`0`是指令中的偏移量部分，表示跳转的相对偏移量。

在执行跳转指令之前，计算出下一条指令的地址。假设当前指令的地址是`A`，则下一条指令的地址为`A + 2`（每条指令的长度为 2 字节）。

使用下一条指令的地址加上偏移量`0`，即`(A + 2) + 0`，得到跳转目标地址。

在执行跳转指令时，程序将转移到计算得到的跳转目标地址。

因此，对于指令`jmpi 0,8`，它将无条件地跳转到当前指令的下一条指令地址加上偏移量`0`的位置，即跳转到下一条指令的地址加上 8 字节的位置。

实验报告

请简述 head.s 的工作原理

当引导 Linux 0.00 内核时，head.s 文件扮演了至关重要的角色，确保系统能够启动和执行操作系统的核心功能。

（1）段寄存器和栈指针的设置：首先，head.s 文件会设置段寄存器和栈指针，以确保程序可以正确地执行和访问内存。这是引导加载程序的一部分，它确保系统处于适当的状态。

（2）读取内核映像文件的头部信息：接下来，head.s 文件将读取内核映像文件的头部信息，以获取内核程序的大小、入口点和其他关键信息。这些信息对于将内核加载到内存的正确位置至关重要。

（3）加载内核程序到内存：此后，head.s 文件的任务是将内核程序从磁盘加载到内存中，将其复制到正确的内存位置，以便系统能够访问它。这确保了内核在内存中正确初始化。

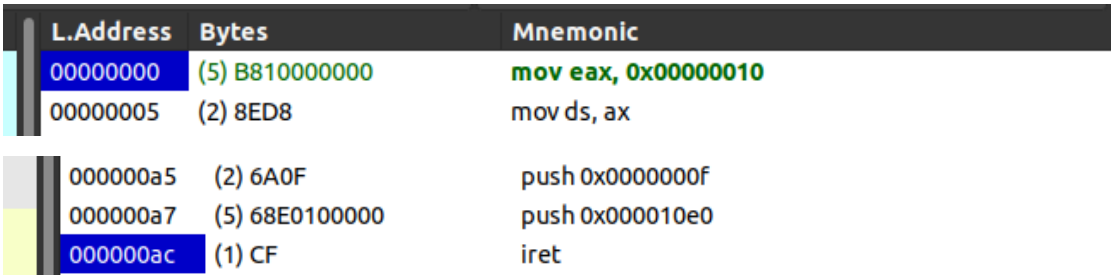
（4）设置系统调用向量表：内核的初始化过程包括设置正确的系统调用向量表，以确保内核程序能够正确响应系统调用。这涉及建立系统调用处理程序的入口点和相关数据结构。

（5）跳转到内核程序的入口点：最后，引导加载程序或内核启动代码将跳转到内核程序

的入口点，将控制权转交给内核。从这一刻开始，内核将开始执行操作系统的核心功能，例如进程管理、内存管理和文件系统操作

请记录 head.s 的内存分布状况，写明每个数据段，代码段，栈段的起始与终止的内存地址

Eg：对于代码段



从 0x00 到 0xac 是 startup_32 的内存分布（与书中的代码做对比即可知道开始和结束位置）；依次找出：

`startup_32`	`0x00`	`0xac`
`setup_gdt`	`0xad`	`0xb4`
`setup_idt`	`0xb5`	`0xc7`
`rp_sidt`	`0xcd`	`0xe4`
`write_char`	`0xe5`	`0x113`
`ignore_int`	`0x114`	`0x129`
`timer_interrupt`	`0x12a`	`0x165`
`system_interrupt`	`0x166`	`0x17c`
`task0`	`0x10e0`	`0x10f3`
`task1`	`0x10f4`	`0x1107`

数据段：

`current`	`0x17d`	`0x180`
`scr_loc`	`0x181`	`0x184`
`lidt_opcode`	`0x186`	`0x18b`
`lgdt_opcode`	`0x18c`	`0x191`
`idt`	`0x198`	`0x997`
`gtd`	`0x998`	`0x9d7`
`ldt0`	`0xbe0`	`0xbf7`
`tss0`	`0xbf8`	`0xc5f`
`ldt1`	`0xc60`	`0xe77`
`tss1`	`0xe78`	`0xedf`

堆栈段：

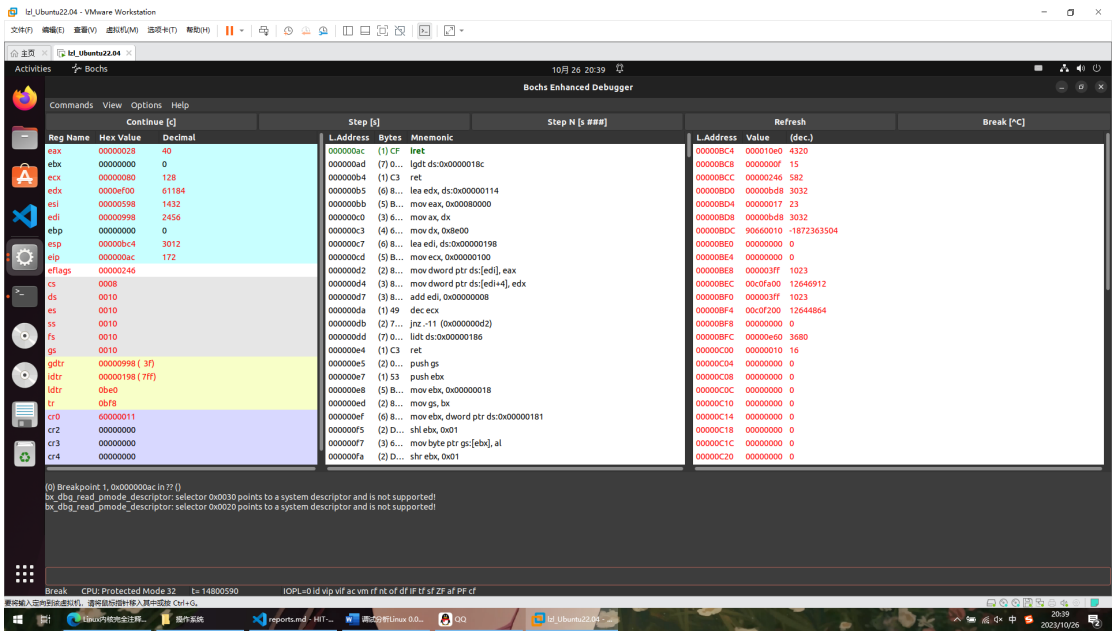
`init_stack`	`0x9d8`	`0xbd7`（栈指针`0xbd8`）
`krn_stk0`	`0xc60`	`0xe5f`（栈指针`0xe60`）
`krn_stk1`	`0xe00`	`0x10df`（栈指针`0x10e0`）
`user_stk1`	`0x1108`	`0x1307`（栈指针`0x1308`）

简述 head.s 57 至 62 行在做什么?

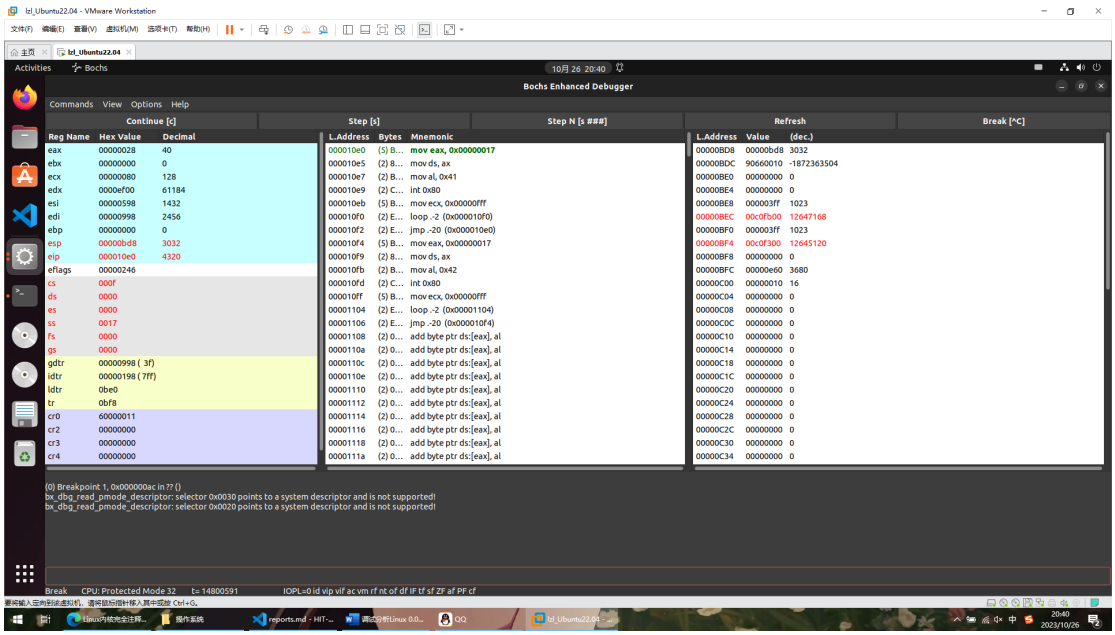
57：把任务 0 当前局部空间数据段（堆栈段）选择符入栈
58：把堆栈指针入栈（也可以把 ESP 入栈）
59：把标志寄存器入栈
60：把当前局部空间代码段选择符入栈
61：把代码指针入栈
62：执行中断返回指令，从而切换到特权级了的任务 0 中执行

简述 iret 执行后， pc 如何找到下一条指令?

在`iret`执行前，`eip`的值为`000000ac`，`esp`为`00000bc4`，`cs`的值为`0008`，栈顶的值为`000010e0`， 然后执行`iret`。



栈内五条内容均被弹出，其中`000010e0`被写入`eip`，同时`0000000f`被写入`cs`，则获得了下一条指令的地址。



记录 iret 执行前后，栈是如何变化的?

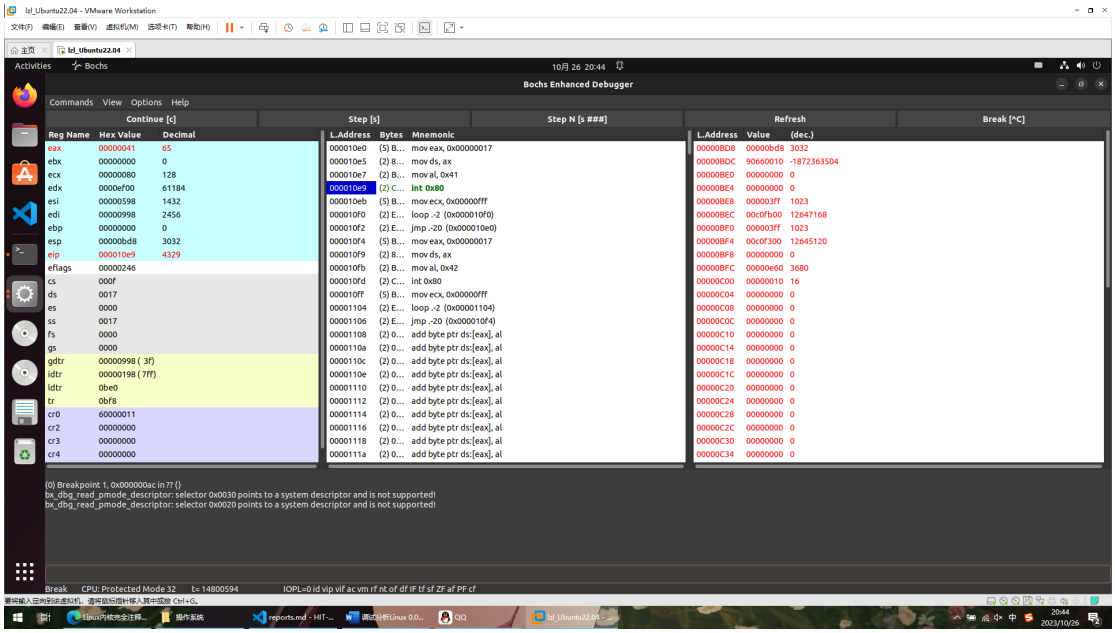
上面两图已经记录:

执行`iret`前，栈顶`SS:ESP`指向`0x10:0x0BC4`，而且还可以发现之前往栈顶里压入的跳转目标地址`0x0F:0x10E0`和用户栈栈顶地址 `0x17:0x0BD8`。

执行'iret'后，栈顶`SS:ESP`自动切换为了之前所指定的用户栈栈顶`0x17:0x0BD8`，而且现在`CS:EIP`跳转到了`0x0F:0x10E0`执行。这就是具有特权级变换的跳转。

当任务进行系统调用时，即 int 0x80 时，记录栈的变化情况。

运行到该语句：现在栈顶`SS:ESP`还是指向`0x17:0x0BD8`，此时`CS:EIP=0x0F:0x10E9`



执行该语句:

此时会发现`SS:ESP`自动切换到了`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（标号为`system_interrupt`）地址`0x08:0x0166`处。并且，可以发现原来的用户栈栈顶地址`0x17:0x0BD8`和中断返回地址`0x0F:0x10EB`被压到了内核栈栈顶。此时`SS:ESP`会自动发生切换的原因是进行带有特权级变换的跳转时，如果是从高特权级跳到低特权级，会将栈顶地址`SS:ESP`切换到`TSS`对应字段指示的值。