1. 调试分析 Linux 0.00 引导程序

1.1. 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；

1.2. 实验内容

1.2.1. 掌握如何手写 Bochs 虚拟机的配置文件

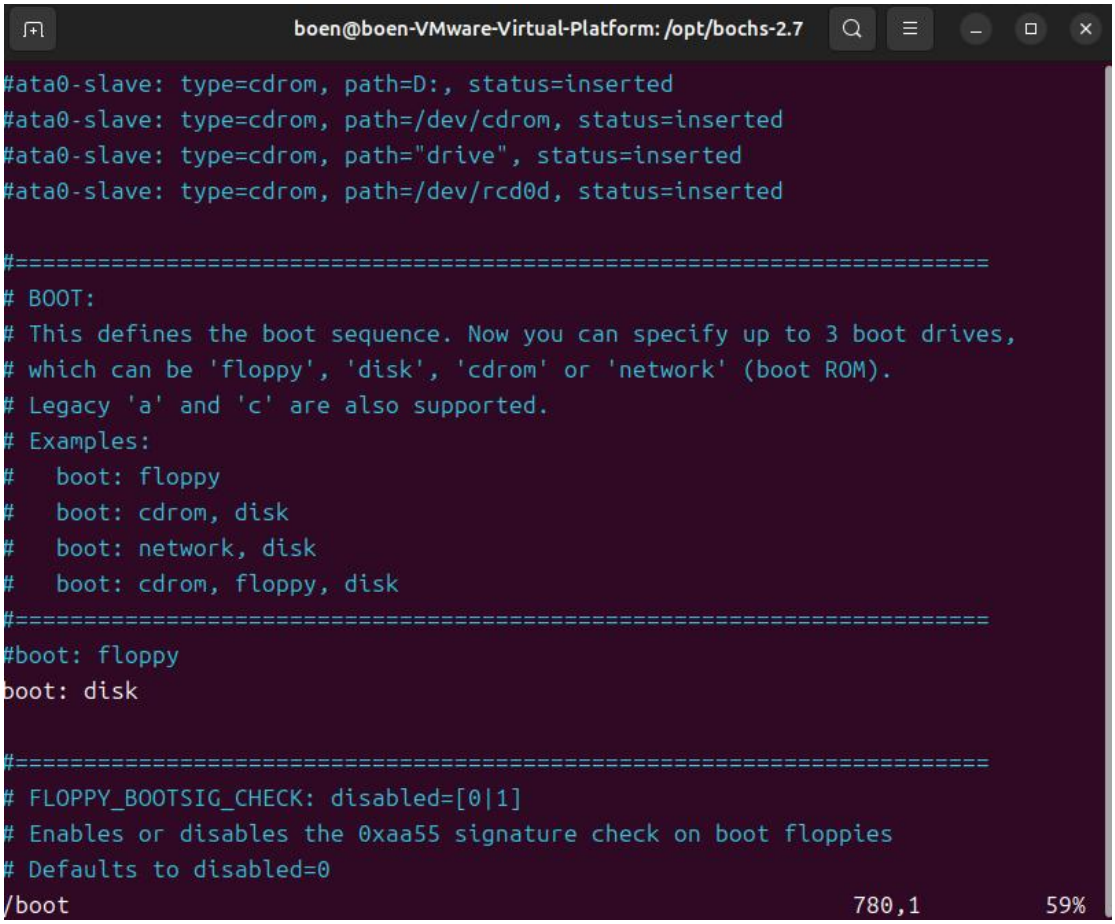
・简介 Bochs 虚拟机的配置文件

Bochs 的配置文件是一个文本文件，通常命名为 .bochsrc,在 Linux 系统重可以 cd 到 bochs 根目录中用 vim 打开编辑，它包含了用于定义虚拟机的各种参数和选项的设置。这些参数和选项允许配置虚拟机的硬件、启动选项、内存分配、设备模拟和其他相关设置。

```
boen@boen-VMware-Virtual-Platform:~/桌面$ cd /opt/bochs-2.7
boen@boen-VMware-Virtual-Platform:/opt/bochs-2.7$ ls -a
.                  config.h                  docs-html      memory
..                 config.h.in               extplugin.h    misc
aclocal.m4         config.log                gdbstub.cc     msrs.def
bios               config.o                  gui            osdep.cc
bochs              config.status             host           osdep.h
bochs.h            config.sub                install-sh     osdep.o
.bochsrc           configure                 instrument     param_names.h
build              configure.in              iodev          PARAM_TREE.txt
bx_debug           .conf.linux               .libs          patches
bxdisasm.cc        .conf.macos               libtool        pc_system.cc
bximage            .conf.macosx              LICENSE        pc_system.h
bxthread.cc        .conf.nothing             logio.cc       pc_system.o
bxthread.h         .conf.sparc               logio.h        plugin.cc
bxthread.o         .conf.win32-cygwin        logio.o        plugin.h
bxversion.h        .conf.win32-vcpp          ltdl-bochs.h   plugin.o
bxversion.h.in     .conf.win64-cross-mingw32 ltdl.c         qemu-queue.h
bxversion.rc       .conf.win64-vcpp          ltdlconf.h     README
bxversion.rc.in    COPYING                   ltdlconf.h.in  README-wxWidgets
CHANGES            cpu                       ltmain.sh      TESTFORM.txt
.conf.amigaos      cpudb.h                   main.cc        TODO
```

・如何设置从软驱启动

打开配置文件，将 floopy 所在行取消注释，同时注释另一行。



・如何设置从硬盘启动

打开配置文件，将 disk 所在行取消注释，同时注释另一行。

・如何设置调试选项

打开配置文件，找到 debug 行，更改相关设置来进行调试，如：

启用调试输出

debug: action="option_name", parameter="value"

设置调试动作为打印所有调试信息

debug: action="debug", option="all"

设置断点

debug: action="bpoint", name="my_breakpoint", type=address, addr=0x1234

action：指定要执行的调试操作。

option：如果存在，可以设置特定的调试选项。

name：为断点指定一个名称，用于标识它。

type：指定断点的类型。

addr：如果设置了地址断点，指定要设置的地址

更改后保存设置，再运行 bochs 即会运行调试好的设置。

```
#=======================================================================
panic: action=ask
error: action=report
info: action=report
debug: action=ignore, pci=report # report BX_DEBUG from module 'pci'

#=======================================================================
# DEBUGGER_LOG:
# Give the path of the log file you'd like Bochs to log debugger output.
# If you really don't want it, make it /dev/null or '-'. :^(
```

1.2.2. 掌握 Bochs 虚拟机的调试技巧

・如何单步跟踪？

输入 s 命令即可单步执行，每次输入 s 命令，虚拟机将执行当前指令并停在下一条指令之前。

・如何设置断点进行调试？

使用 b +地址命令设置断点，再点击运行，bochs 就会在断点位置停止

・如何查看通用寄存器的值？

调试界面左边蓝色部分：

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？

调试界面左侧紫色部分：

cr0	60000010
cr2	00000000
cr3	00000000
cr4	00000000
efer	00000000

・如何查看内存指定位置的值？

使用如下命令（x）：

x /<count><format> <address>

<count>：指定要查看的数据项数量。

<format>：指定数据的显示格式，如 b（字节）、w（字）、d（双字）等。

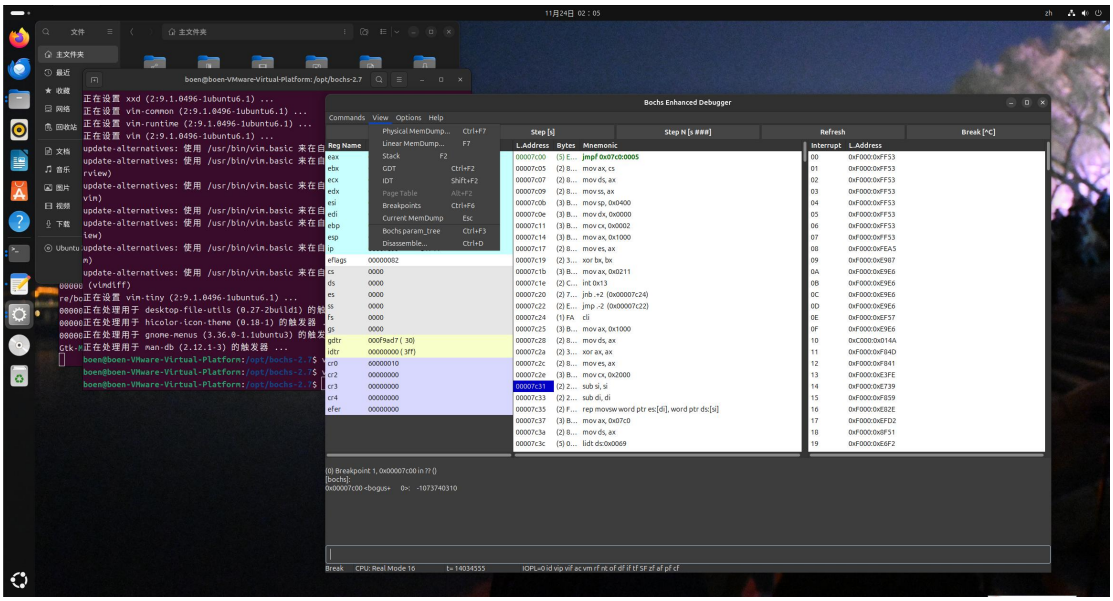
<address>：指定要查看的内存地址。

X /1wd 0x7c00:

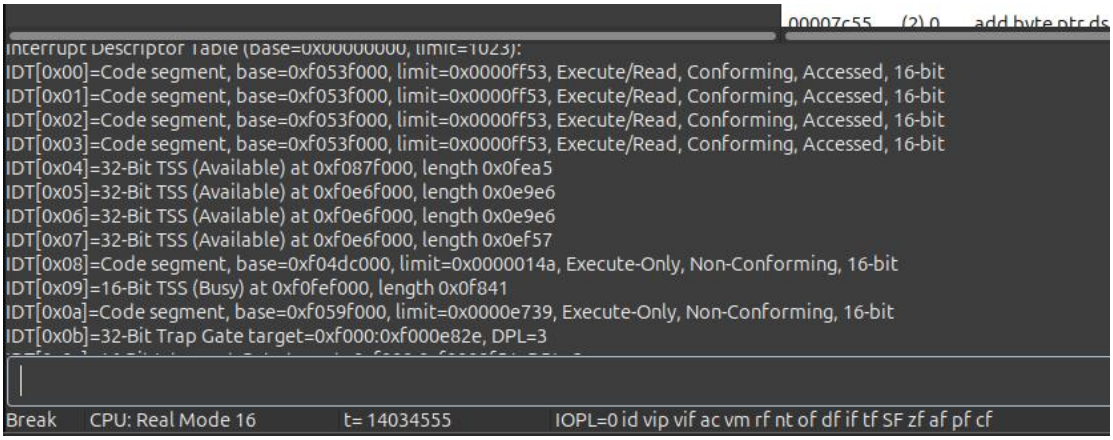
```
[bochs]:
0x00007c00 <bogus+     0>:  -1073740310
```

· 如何查看各种表，如 gdt，idt，ldt 等？

点击调试界面左上角的 view，即可选择查看 GDT， IDT， STACK 等，打开后其内容显示在调试器右边部分的界面，LDT， TSS 可以从 GDT 中得到基址间接查看。

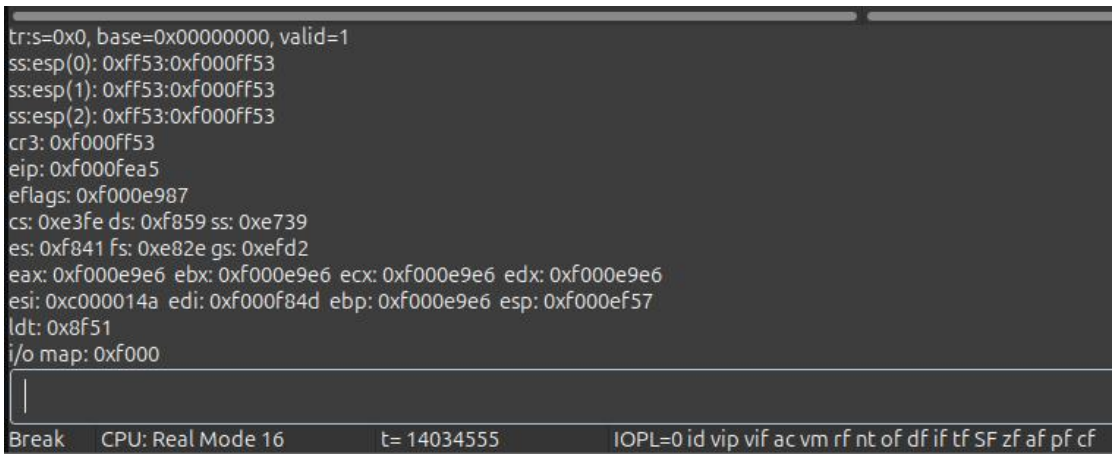


可以使用 info+表名查看表中的内容（info idt（小写））：



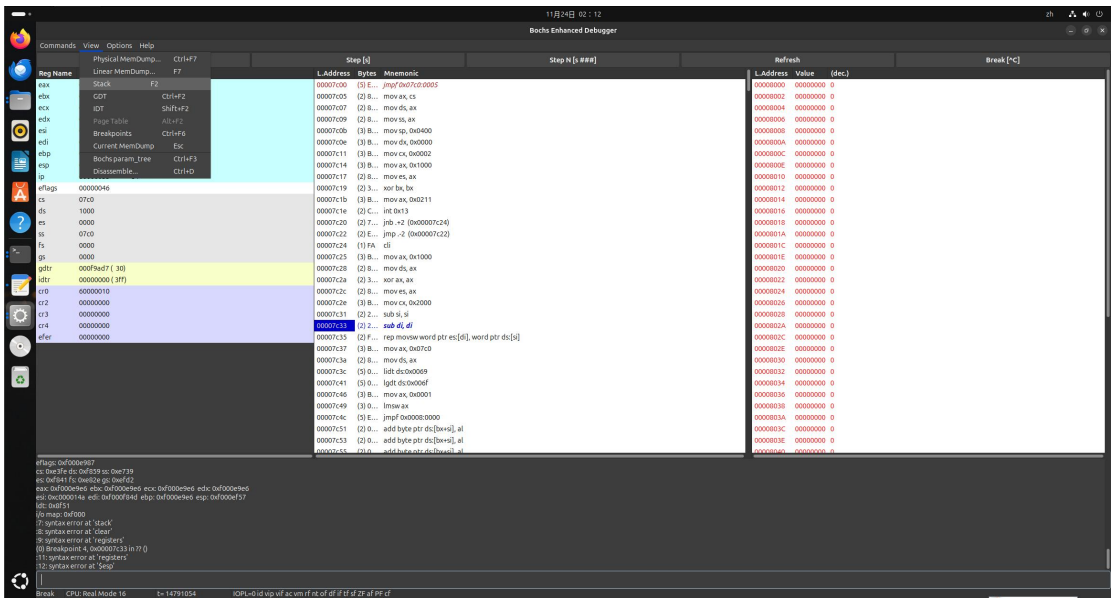
· 如何查看 TSS？

同上 info tss（小写）：



· 如何查看栈中的内容？

同同上上，在断点时，使用 view 中的 stack 查看：



· 如何在内存指定地方进行反汇编？

使用 disasm 命令：

disasm <address> <count>

<address>：指定要反汇编的内存地址。

<count>：指定要反汇编的指令数量。

disasm 0x1234 10:

1.2.3. 计算机引导程序

・如何查看 0x7c00 处被装载了什么？

可以用 x 命令查看，如 x /100wd 0x7c00,或在其位置设置一个断点查看，但是又课上老师讲授可知装在的是 head 的内容：

```
[bochs]:
0x00007c00 <bogus+   0>:  -1073740310  -1899459577  -1127182632  12190720
0x00007c10 <bogus+  16>:  178432    -1911553864  -1193594432  332202513
0x00007c20 <bogus+  32>:  -18152845 268482810    -1070475122  12173454
0x00007c30 <bogus+  48>:  703998240    -1197083649  -661780544   1763574031
0x00007c40 <bogus+  64>:  369168128    28835951  -268366080    134217962
0x00007c50 <bogus+  80>:  0   0   524032    -1063649280
0x00007c60 <bogus+  96>:  524032    -1064173568  0   -16777216
0x00007c70 <bogus+ 112>: 8147207   0   0   0
0x00007c80 <bogus+ 128>: 0   0   0   0
0x00007c90 <bogus+ 144>: 0   0   0   0
```

```
00007c00    (5) E...  jmpf 0x07c0:0005
```

・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;

利用引导启动程序 boot.s，boot.s 主要功能就是将软盘或映像文件中的 head 内核 diamagnetic 加载到内存中某个指定位置处，并在设置临时 GDT 表等信息后，把处 理器设置成运行在保护模式下，然后跳转到 hed 代码处去运行代码。

实际上，boot.s 程序会首先利用 ROM BIOS 中断 int 0x13 把软盘中的 head 代码读入到内存 0x10000 位置开始处，然后再把这段 head 代码移动到内存 0 处，最后设置控制寄存器 CR0 中的开启保护运行模式，并跳转到内存 0 处开始执行 head 代码。

・如何查看实模式的中断程序？

查找中断向量表：在实模式下，中断处理程序通常通过中断向量表来查找。这个表包含中断号与中断处理程序的关联关系。

查看中断处理程序：一旦找到了中断处理程序的地址，就可以利用 x addr 指令即可查看对应地址的内存内容，即中断程序的内容。

・如何静态创建 gdt 与 idt ？

创建 GDT 和 IDT 表项：定义 GDT 和 IDT 表项的结构。每个表项包含段描述符或中断描述符的相关信息，例如段基址、段限制、访问权限等。根据需要，创建所有所需的表项。

初始化 GDT 和 IDT 表项：为每个表项设置适当的值，包括段的起始地址、限制、特权级别、类型（代码段、数据段、中断门等）等。

创建 GDT 和 IDT 表：将所有初始化的表项组合成 GDT 和 IDT 表，这些表通常存储在内存中。

加载 GDT 和 IDT：使用汇编代码将 GDT 和 IDT 表的地址加载到处理器的 GDTR 和 IDTR 寄存器中。这通常涉及到汇编指令 lgdt（加载 GDT）和 lidt（加载 IDT）。

启用中断：如果正在设置 IDT 以处理中断，确保启用中断处理器，以便它可以响应中断。这可以通过设置处理器的中断标志位（IF）来完成。

编写中断处理程序：如果设置了 IDT 来处理中断，需要编写相应的中断处理程序。

汇编和链接：将所有汇编代码和数据结构汇编并链接成可执行文件。这个文件将包含 GDT 和 IDT 表的初始化和加载代码，以及任何必要的中断处理程序。

加载到目标系统：将生成的可执行文件加载到目标系统的内存中，并执行以初始化 GDT 和 IDT 表。

・如何从实模式切换到保护模式？

准备 GDT 和 IDT：在保护模式下，需要配置全局描述符表（GDT）和中断描述符表（IDT）。需要创建和初始化这些表，包括定义段描述符和中断门。

加载 GDT 和 IDT：使用 lgdt 汇编指令加载 GDT 表的地址到 GDTR 寄存器，并使用 lidt 指令加载 IDT 表的地址到 IDTR 寄存器。

设置 CR0 寄存器：将控制寄存器 CR0 的第 0 位（PE 位）设置为 1，以启用保护模式。这可以通过执行汇编指令 mov eax, cr0，or eax, 1，mov cr0, eax 来完成。

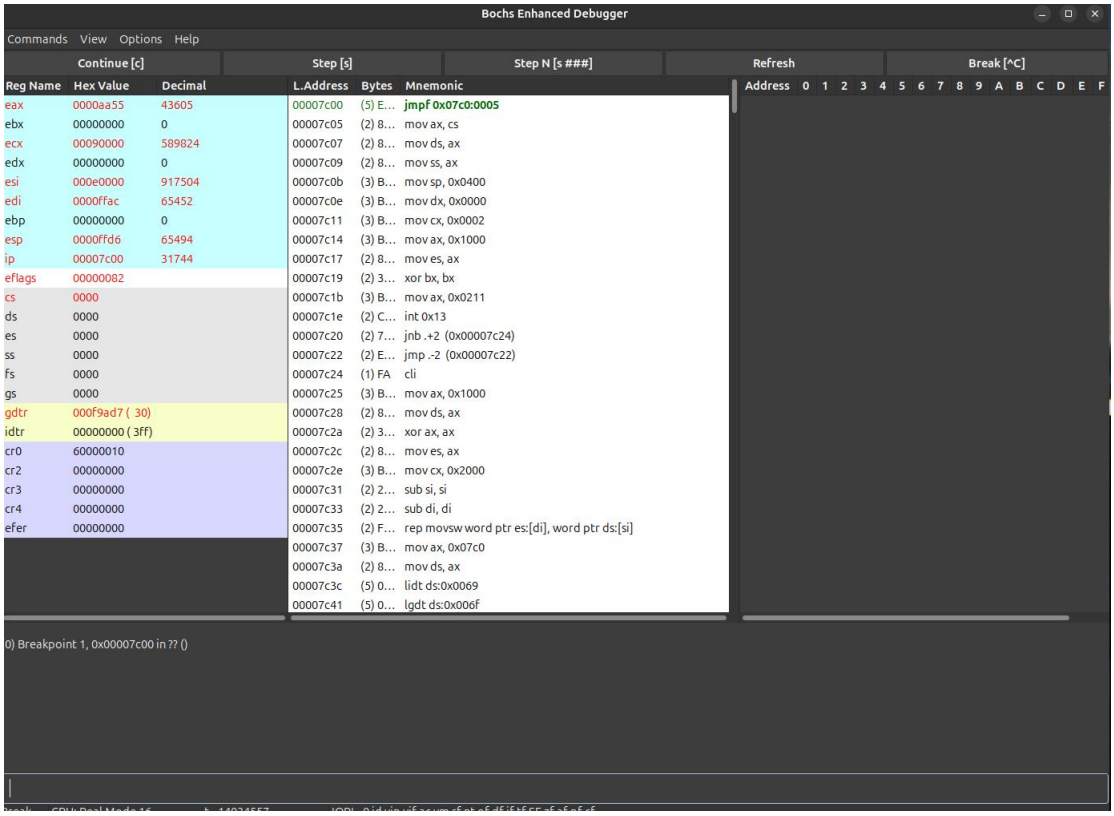
跳转到新代码段：在切换到保护模式后，执行 far jump 指令以跳转到新的代码段。通常，会跳转到新的代码段以开始执行保护模式下的代码。

设置栈指针：在保护模式下，通常需要重新设置栈指针（SP）以指向新的栈段。这是因为在实模式下，栈通常在段 0x0000 下，而在保护模式下通常会有不同的栈段。

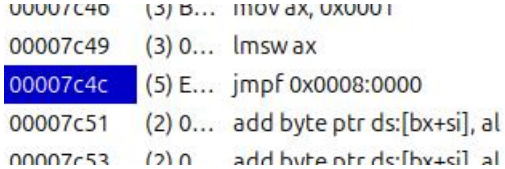
编写保护模式代码：一旦切换到保护模式，需要编写适用于该模式的代码。

・调试跟踪 jmpi 0,8，解释如何寻址？

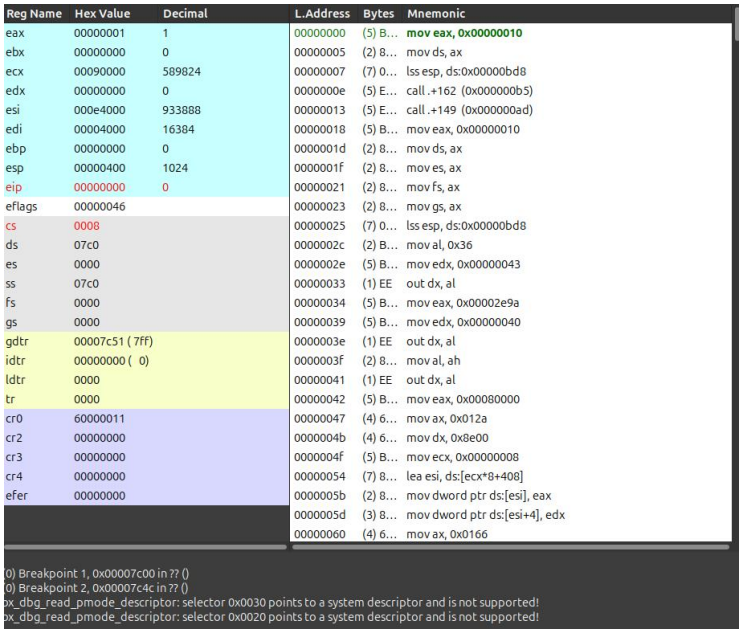
首先在 0x7c00 处设置断点，运行程序，查看 boot.s 的汇编代码：



找到 jumpi 0 8 位置，设置一个断点



Continue 到断点后 step 1 向前运行一步发现跳转到了 0x0000：



如果当前代码段的基址是 0x0000，偏移地址为 0x0008，那么目标地址将是 0x0000 + 0x0008 = 0x0008。

jmpi 指令执行后，控制权将转移到新代码段的指定地址，开始执行那里的指令。

1.3. 实验报告

通过仔细的调试与跟踪程序，完成以下任务：

・请简述 head.s 的工作原理

通常情况下，head.s 是引导加载程序（bootloader）的一部分，用于引导加载操作系统内核，包含 32 位保护模式初始化设置代码、时钟中断代码、系统调用中断代码和两个任务的代码。

加载到内存：head.s 是一个汇编源代码文件，经过汇编和链接后，生成二进制可执行文件，通常是一个引导扇区（boot sector）。该文件必须存储在引导设备（如硬盘或软驱）的引导扇区中。

引导加载程序：计算机启动时，处理器会加载引导设备的引导扇区（通常位于磁盘的第一个扇区）到内存地址 0x7C00。这个扇区通常包含了 head.s 的代码。

设置环境：head.s 开始执行，它的主要任务是设置一个适当的环境，以准备加载操作系统内核。这通常涉及到以下几个步骤：

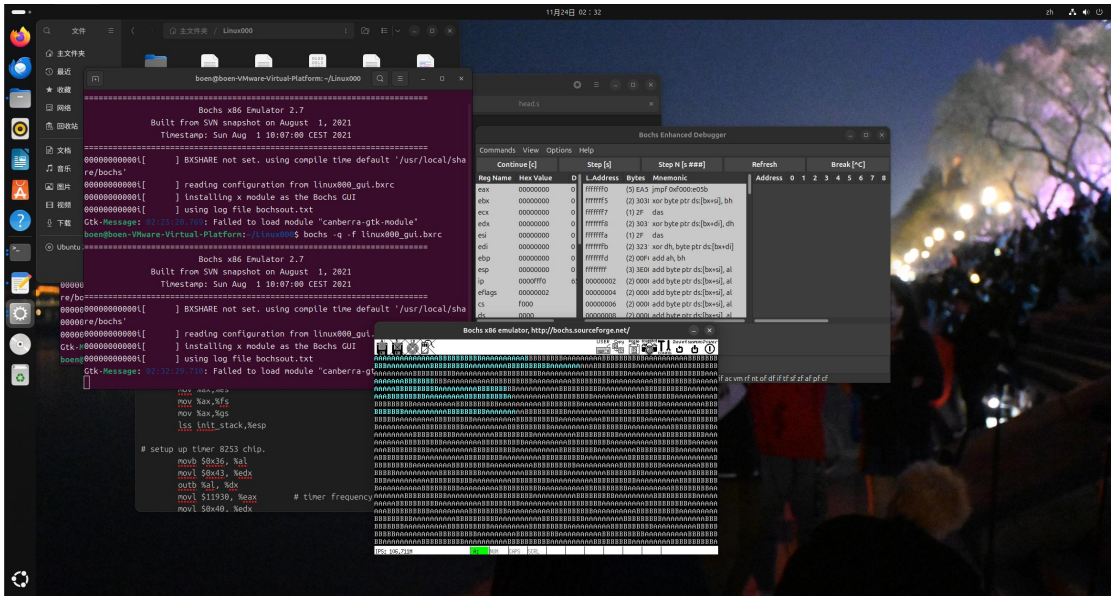
初始化 GDT/IDT

设置系统定时器芯片 8253

初始化 TSS

跳转到 task0 的用户态程序

task0 或 task1 的用户态程序在运行时，通过系统调用 int 0x80 向屏幕上打印字符 A 或 B；时钟中断发生时，内核的中断处理程序实现 task0 和 task1 的任务切换，运行截图：



初始化 GDT/IDT

设置 GDT/IDT 代码如下，调用了两个子程序 setup_gdt 和 setup_idt（LInux 上没有 VS 所以用主机下了一份用来截图）：

```
# setup base fields of descriptors.
    call setup_idt
    call setup_gdt
    movl $0x10,%eax     # reload all the segment registers
    mov %ax,%ds     # after changing gdt.
    mov %ax,%es
    mov %ax,%fs
    mov %ax,%gs
    lss init_stack,%esp
```

```
setup_gdt:
    lgdt lgdt_opcode
    ret

setup_idt:
    lea ignore_int,%edx
    movl $0x00080000,%eax
    movw %dx,%ax            /* selector = 0x0008 = cs */
    movw $0x8E00,%dx        /* interrupt gate - dpl=0, present */
    lea idt,%edi
    mov $256,%ecx
```

设置系统定时器芯片：

```
# setup up timer 8253 chip.
    movb $0x36, %al# 控制字：设置通道 0 工作在方式 3、计数初值采用二进制。
    movl $0x43, %edx# 8253 芯片控制字寄存器写端口。
    outb %al, %dx
    movl $11930, %eax        # timer frequency 100 HZ
    movl $0x40, %edx# 通道 0 的端口。
    outb %al, %dx # 分两次把初始计数值写入通道 0。
    movb %ah, %al
    outb %al, %dx
```

跳转到 task 0：

```
# Move to user mode (task 0)
    pushfl                  # 将EFLAGS压栈
    andl $0xffffbfff, (%esp) # EFLAGS 的NT 标志位（第14 位）置0
    popfl
    movl $TSS0_SEL, %eax     # 把任务0的TSS段选择符加载到TR
    ltr %ax
    movl $LDT0_SEL, %eax     # 把任务0的LDT加载到LDTR
    lldt %ax
    movl $0, current         # 任务号
    sti
    pushl $0x17              # 数据段选择符
    pushl $init_stack        # 栈指针
    pushfl                   # EFLAGS
    pushl $0x0f              # 代码段选择符
    pushl $task0             # task0程序入口
    iret
```

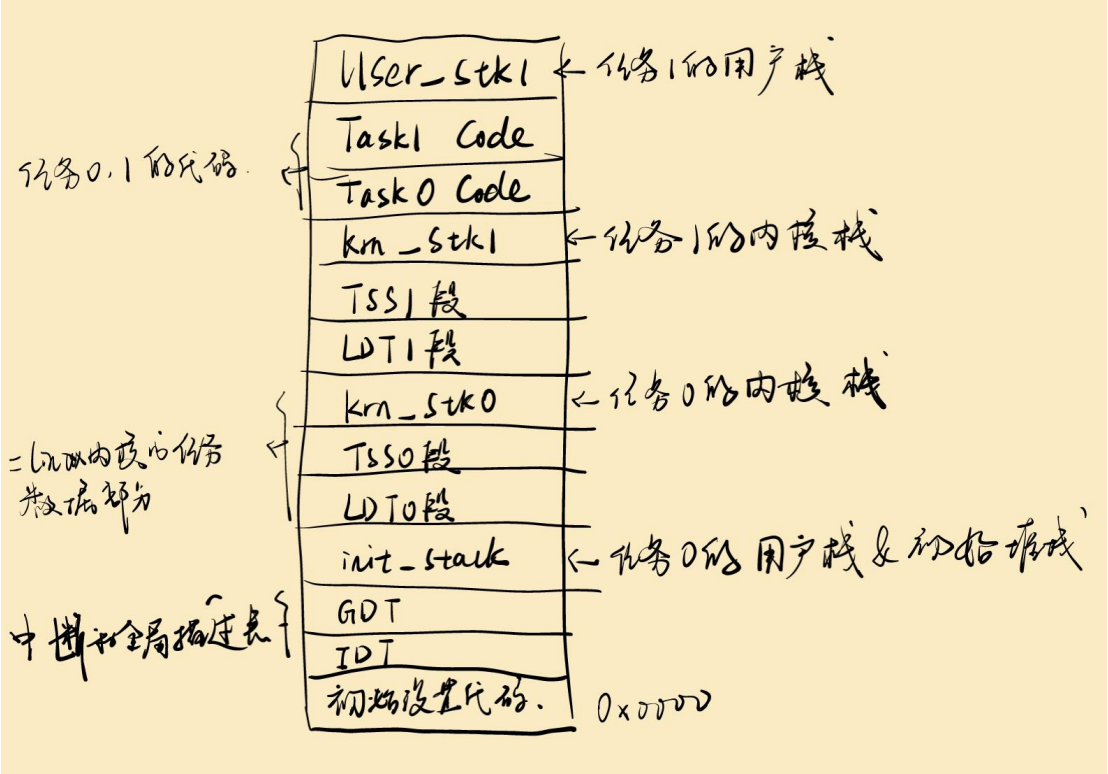
最后执行 iret 时，iret 会把栈顶弹出，更新 CS 和 IP，然后把下一个栈顶弹出，更新 EFLAGS，然后把下两个栈顶弹出，更新 SS 和 ESP，此时便跳转到 task0 下执行。

task0 代码：

```
task0:
    movl $0x17, %eax
    movw %ax, %ds
    movb $65, %al              /* print 'A' */
    int $0x80
    movl $0xfff, %ecx
1:  loop 1b
    jmp task0
```

task0 和 task1 代码唯一区别就是传入的字符不同。它们都使用 int 80 系统调用打印字符，然后进行 4095 次的空循环。

・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。



1） 栈段

名称	起始地址	终止地址
init_stack	0x9d8	0xbd8
krn_stk0	0xc60	0xe60
krn_stk1	0xee0	0x10e0
usr_stk1	0x1108	0x1308

2）数据段

名称	起始地址	终止地址
current	0x17d	0x180
scr_loc	0x181	0x184
lidt_opcode	0x186	0x18b
lgdt_opcode	0x18c	0x191
idt	0x198	0x997
gdt	0x998	0x9d7

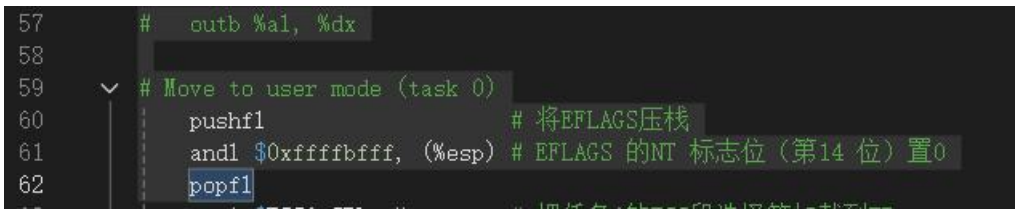
名称	起始地址	终止地址
ldt0	0xbe0	0xbf7
tss0	0xbf8	0xc5f
ldt1	0xc60	0xe77
tss1	0xe78	0xedf

3）代码段

名称	起始地址	终止地址
startup_32	0x00	0xac
setup_gdt	0xad	0xb4
setup_idt	0xb5	0xe4
write_char	0xe5	0x113
ignore_int	0x114	0x129
timer_interrupt	0x12b	0x165
system_interrupt	0x166	0x17c
task0	0x10e0	0x10f3
task1	0x10f4	0x1107

- 简述 head.s 57 至 62 行在做什么？

用 vs 打开偷瞄一眼，很明显不是需要的字段，于是开始调试：



首先需要找到 57-62 行的位置，所以先在第 56 行设置断点，然后对 57-62 行进行反汇编（即 disasm 56address 7）出现这些东西（为什么命令区不能放大？）：

```
(0) Breakpoint 2, 0x0000009c in ?? ()
00000007: (              ): lss esp, ds:0x00000bd8   ; 0fb225d80b0000
0000000e: (              ): call .+162  (0x000000b5) ; e8a2000000
00000013: (              ): call .+149  (0x000000ad) ; e895000000
00000018: (              ): mov eax, 0x00000010     ; b810000000
0000001d: (              ): mov ds, ax              ; 8ed8
0000001f: (              ): mov es, ax              ; 8ec0
00000021: (              ): mov fs, ax              ; 8ee0
00000023: (              ): mov gs, ax              ; 8ee8
00000025: (              ): lss esp, ds:0x00000bd8   ; 0fb225d80b0000
0000002c: (              ): mov al, 0x36            ; b036
0000002e: (              ): mov edx, 0x00000043     ; ba43000000
00000033: (              ): out dx, al              ; ee
```

```
00000034: (              ): mov eax, 0x00002e9a     ; b89a2e0000
00000039: (              ): mov edx, 0x00000040     ; ba40000000
0000003e: (              ): out dx, al              ; ee
0000003f: (              ): mov al, ah              ; 88e0
00000041: (              ): out dx, al              ; ee
00000042: (              ): mov eax, 0x00080000     ; b800000800
00000047: (              ): mov ax, 0x012a          ; 66b82a01
0000004b: (              ): mov dx, 0x8e00          ; 66ba008e
0000004f: (              ): mov ecx, 0x00000008     ; b908000000
00000054: (              ): lea esi, ds:[ecx*8+408]  ; 8d34cd98010000
0000005b: (              ): mov dword ptr ds:[esi], eax ; 8906
0000005d: (              ): mov dword ptr ds:[esi+4], edx ; 895604
00000060: (              ): mov ax, 0x0166          ; 66b86601
```

```
00000064: (              ): mov dx, 0xef00          ; 66ba00ef
00000068: (              ): mov ecx, 0x00000080     ; b980000000
0000006d: (              ): lea esi, ds:[ecx*8+408]  ; 8d34cd98010000
00000074: (              ): mov dword ptr ds:[esi], eax ; 8906
00000076: (              ): mov dword ptr ds:[esi+4], edx ; 895604
00000079: (              ): pushf                   ; 9c
```

```
0000007a: (              ): and dword ptr ss:[esp], 0xffffbfff ; 812424ffbfffff
00000081: (              ): popf                    ; 9d
00000082: (              ): mov eax, 0x00000020     ; b820000000
00000087: (              ): ltr ax                  ; 0f00d8
0000008a: (              ): mov eax, 0x00000028     ; b828000000
0000008f: (              ): lldt ax                 ; 0f00d0
00000092: (              ): mov dword ptr ds:0x0000017d, 0x00000000 ; c7057d01000000000000
```

同样的，在源文件中找到这些行的内容：

```
pushl $0x17
pushl $init_stack
pushfl
pushl $0x0f
pushl $task0
iret
```

分析获得如下结论：

pushl $0x17	把任务 0 当前局部空间数据段（堆栈段）选择符入栈
pushl $init_stack	把堆栈指针入栈（也可以直接把 ESP 入栈）
pushfl	把标志寄存器值入栈
pushl $0x0f	把当前局部空间代码段选择符入栈

pushl $task0	把代码指针入栈
iret	执行中断返回指令，从而切换到特权级 3 的任务 0 中执行

整体上，该段代码为任务 0 的切换做准备。它将任务 0 的局部数据段选择子、堆栈指针、标志寄存器值、局部代码段选择子和入口地址压入堆栈。然后通过 IRET 指令，切换到任务 0 的特权级，开始执行任务 0 的代码，实现了任务的切换。

·简述 iret 执行后， pc 如何找到下一条指令？

恢复 IP（指令指针寄存器）状态： IRET 指令会从堆栈中弹出保存的 IP 值（中断发生前的指令指针），并将该值存入 IP 寄存器。

恢复 CS（代码段寄存器）状态： IRET 指令会从堆栈中弹出保存的 CS 值（中断发生前的代码段选择子），并将该值存入 CS 寄存器。

恢复标志寄存器状态： IRET 指令会从堆栈中弹出保存的标志寄存器（EFLAGS）的值，并将该值存入标志寄存器。

恢复 ESP（堆栈指针寄存器）状态： IRET 指令会从堆栈中弹出保存的 ESP 值（中断发生前的堆栈指针），并将该值存入 ESP 寄存器。

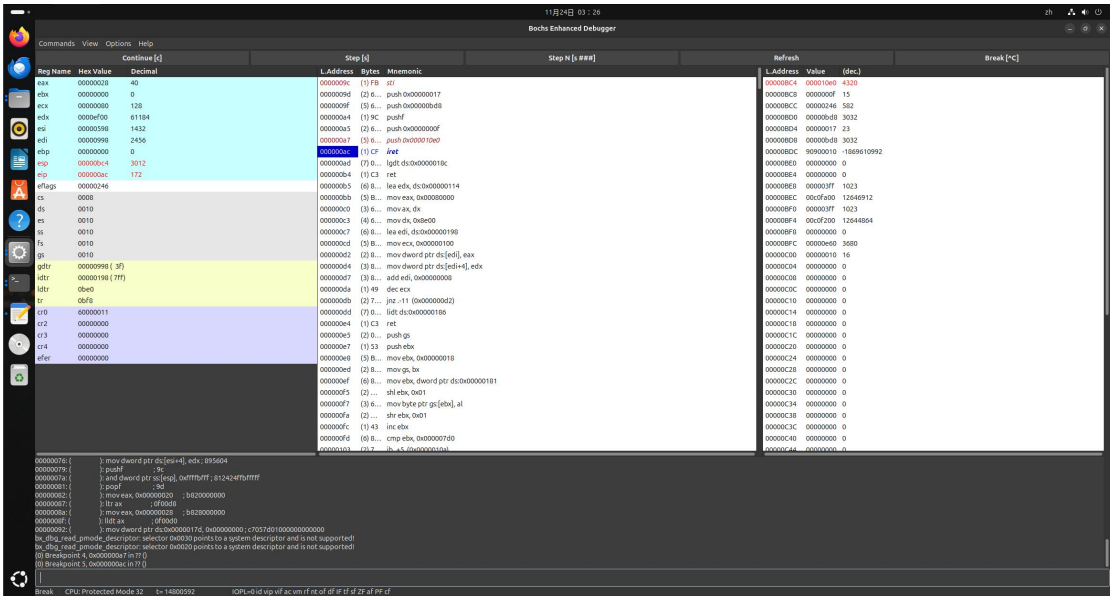
恢复 SS（堆栈段寄存器）状态： IRET 指令会从堆栈中弹出保存的 SS 值（中断发生前的堆栈段选择子），并将该值存入 SS 寄存器。

在程序执行时可能会触发中断，导致进入中断处理函数。在进入中断处理函数之前，当前程序的状态被保存到堆栈中。当中断处理函数执行 IRET 指令时，它从堆栈中恢复之前保存的状态信息，包括指令指针、代码段选择子、标志寄存器等。通过这些恢复的信息，程序回到中断发生前的状态，并继续执行导致中断的指令的下一条指令。这个过程中，程序计数器（PC）找到了下一条指令的执行地址。

·记录 iret 执行前后，栈是如何变化的？

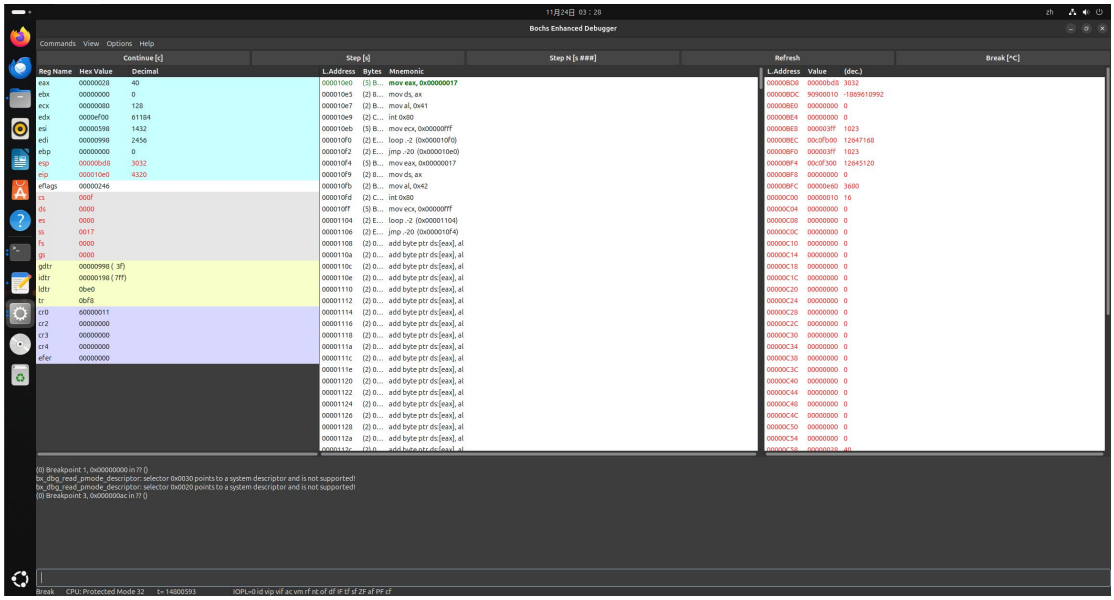
在 iret 位置设置断点，并且 continue 检查前后：

执行前：



0x0BC4-0x0BD4 地址，栈内的内容就是 57 至 61 行代码执行时压入栈内的内容，与指令对应

执行后：



0x0BC4-0x0BD4 地址，栈内的内容都已被弹出，栈为空，最上方的为栈底(0x0BD8)的内容。

iret 指令会弹出之前被压入栈的值，以恢复任务 0 的状态。

iret 弹出代码指针（EIP）的值，指示了下一条要执行的指令地址。

然后它弹出代码段选择符（CS），指示了代码段的位置。

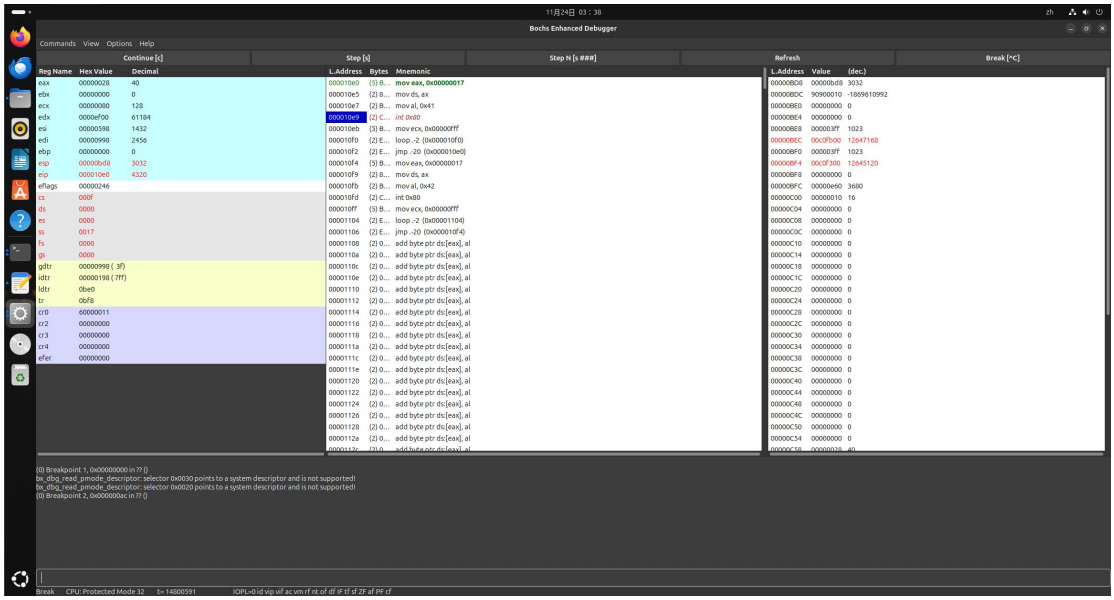
接着，它弹出标志寄存器（EFLAGS）的值，以恢复标志状态。

最后，它弹出堆栈指针（ESP）的值，以确保栈指针正确指向下一个栈帧。

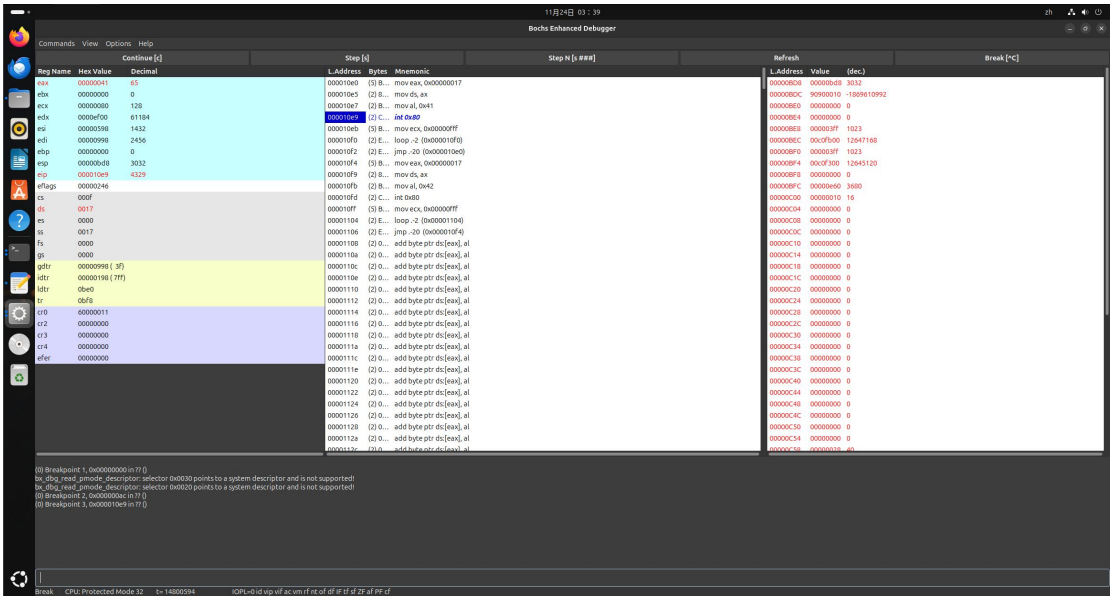
iret 将控制传递到任务 0 中的下一条指令，以继续程序的正常执行。

- 当任务进行系统调用时，即 int 0x80 时，记录栈的变化情况。

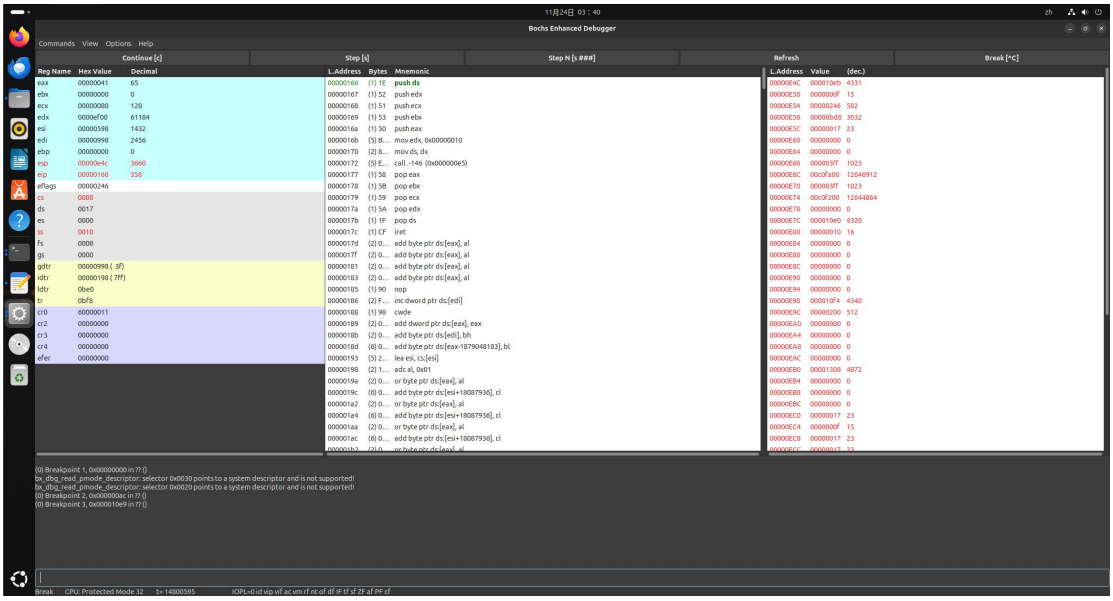
跟踪 iret 的执行发现 int0x80 指令：



设置断点，记录到 0x80 执行之前：



当任务 0 希望执行系统调用时，它会将系统调用号和相关参数加载到寄存器中。
寄存器中的内容，包括系统调用号和参数，通常在进入内核前被保存在寄存器中。
栈上包含了任务 0 正常执行的栈帧，包括函数调用的参数、局部变量等。
单步运行，查看执行之后的情况：



当任务 0 触发 int 0x80 指令时，处理器会执行以下操作：
压入标志寄存器（EFLAGS）的值。
压入代码段选择符（CS）的值。
压入返回地址，指向系统调用处理程序。
压入系统调用号和参数。
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，它会将返回值存储在一个特定的寄存器中，通常是 EAX 寄存器。
处理程序使用 iret 指令返回到用户态，这会将栈上的内容弹出，恢复到 int $0x80 指令执行前的状态。

1.3.1. 评分标准

记录描述要详细完整，每题 15%，总共 90%
格式规范美观，10%