###### 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；

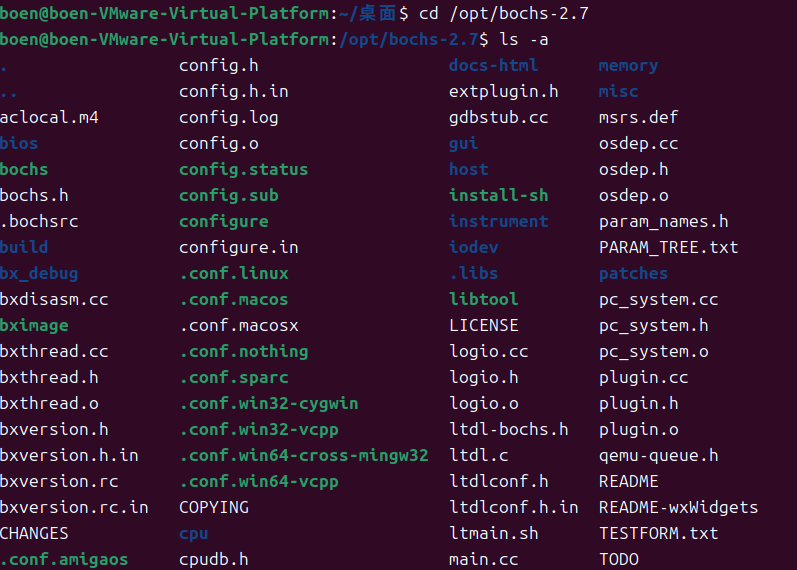
掌握操作系统启动的步骤；

1.2. 实验内容

1.2.1. 掌握如何手写Bochs虚拟机的配置文件

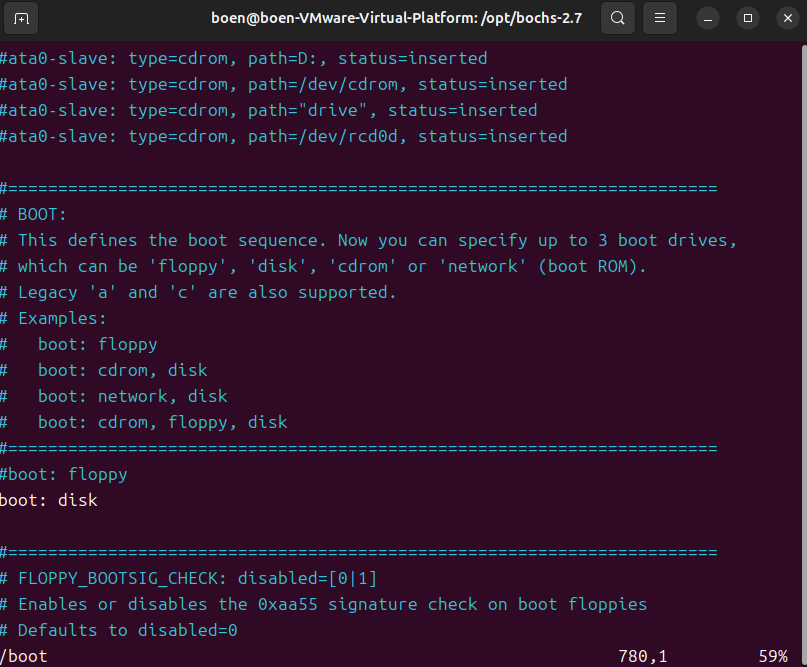
·简介 Bochs 虚拟机的配置文件

 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。



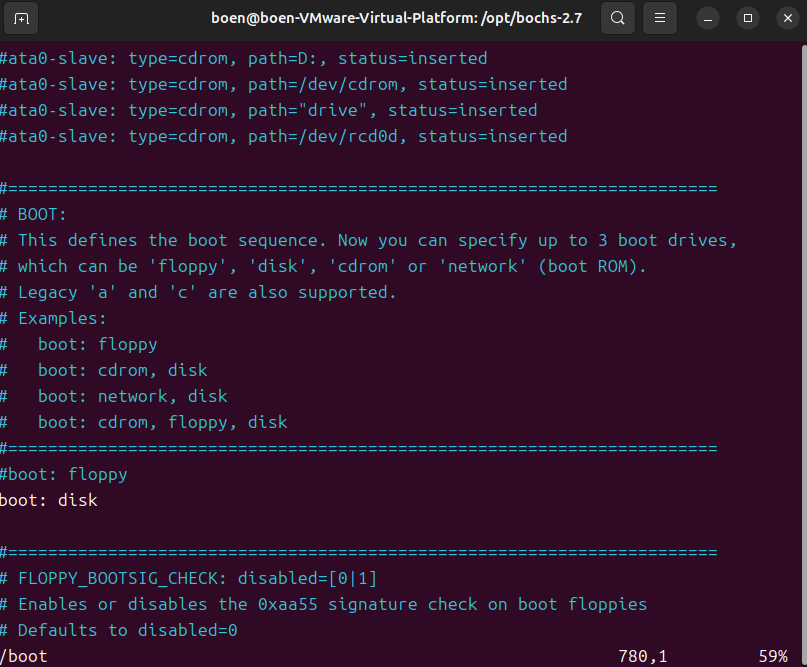
·如何设置从软驱启动

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。

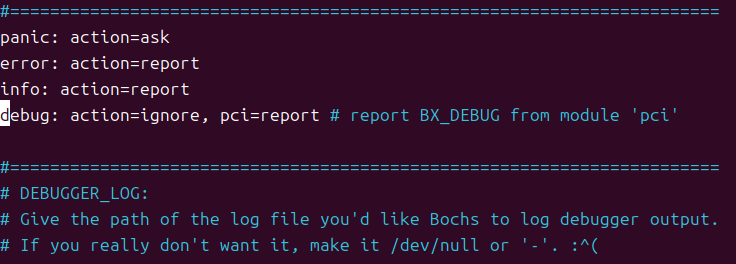
option：如果存在，可以设置特定的调试选项。

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。



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：



·如何查看系统寄存器的值？

调试界面左侧紫色部分：



·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。

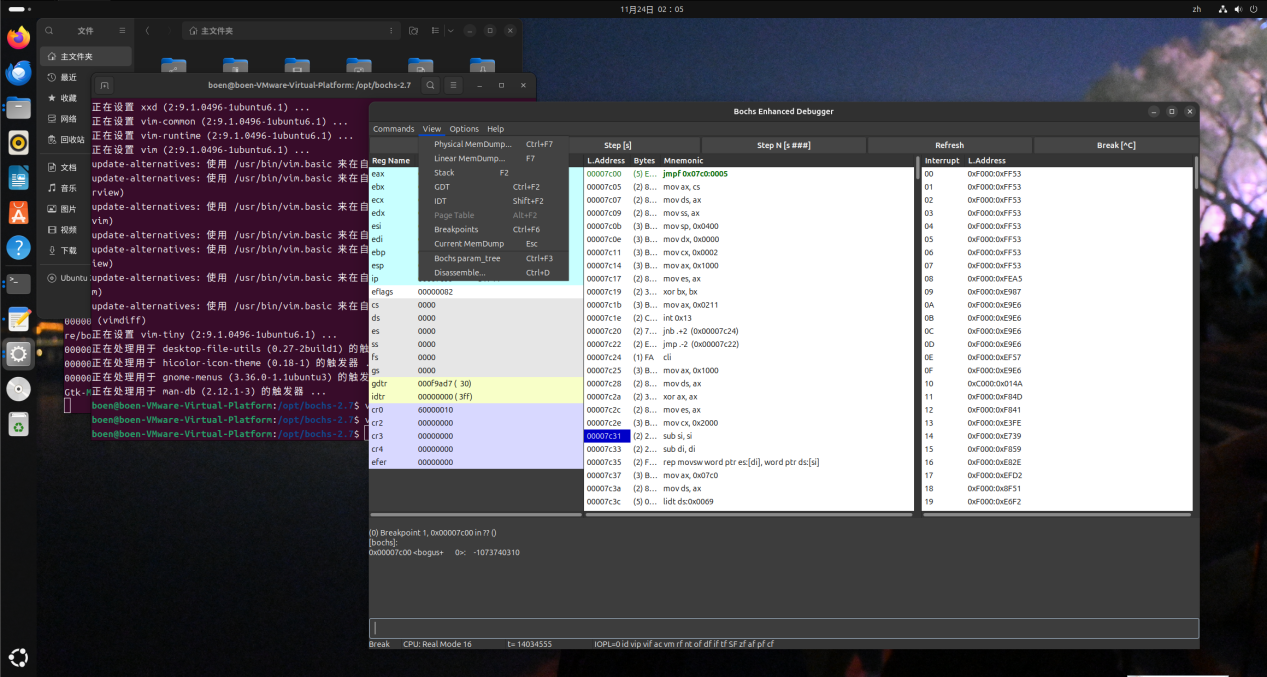
# <format>：指定数据的显示格式，如 b（字节）、w（字）、d（双字）等。

# <address>：指定要查看的内存地址。

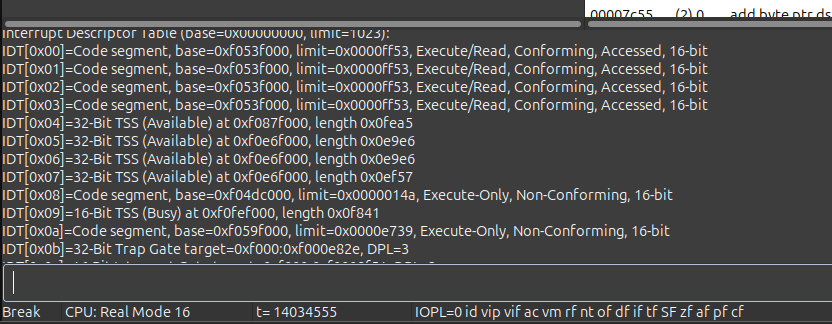
X /1wd 0x7c00:  
 18b4061fdbefcfeac008bbb6c204641

·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。

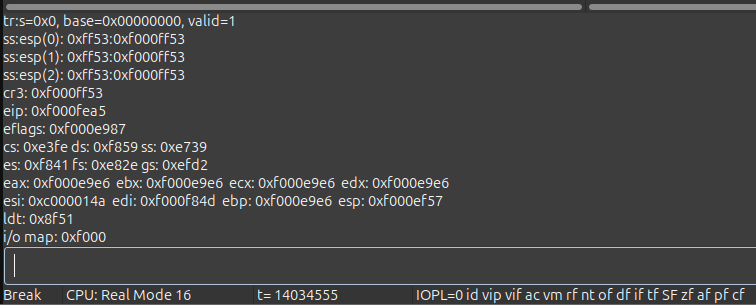


可以使用info+表名查看表中的内容（info idt（小写））：



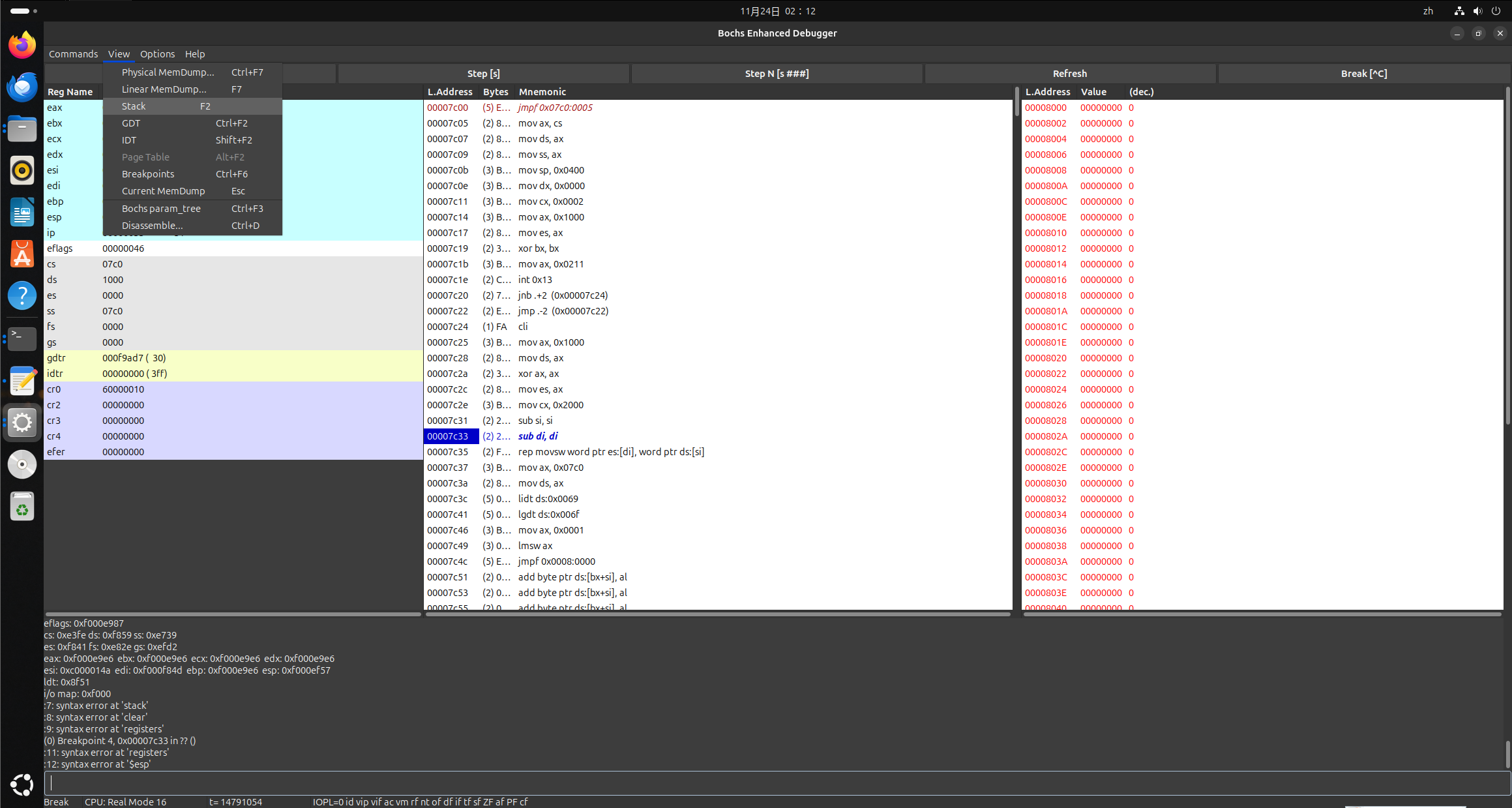
·如何查看 TSS？

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：



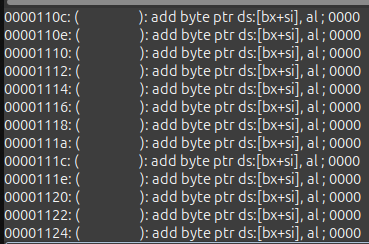
·如何在内存指定地方进行反汇编？

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。

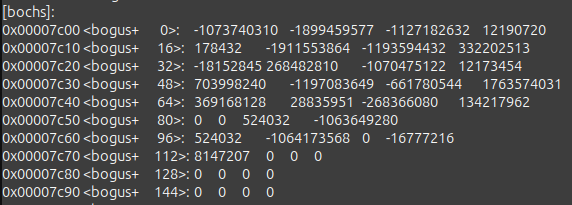
disasm 0x1234 10:



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：



e0994d377c73dd2b6b1dc7b5a079dbb

·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。

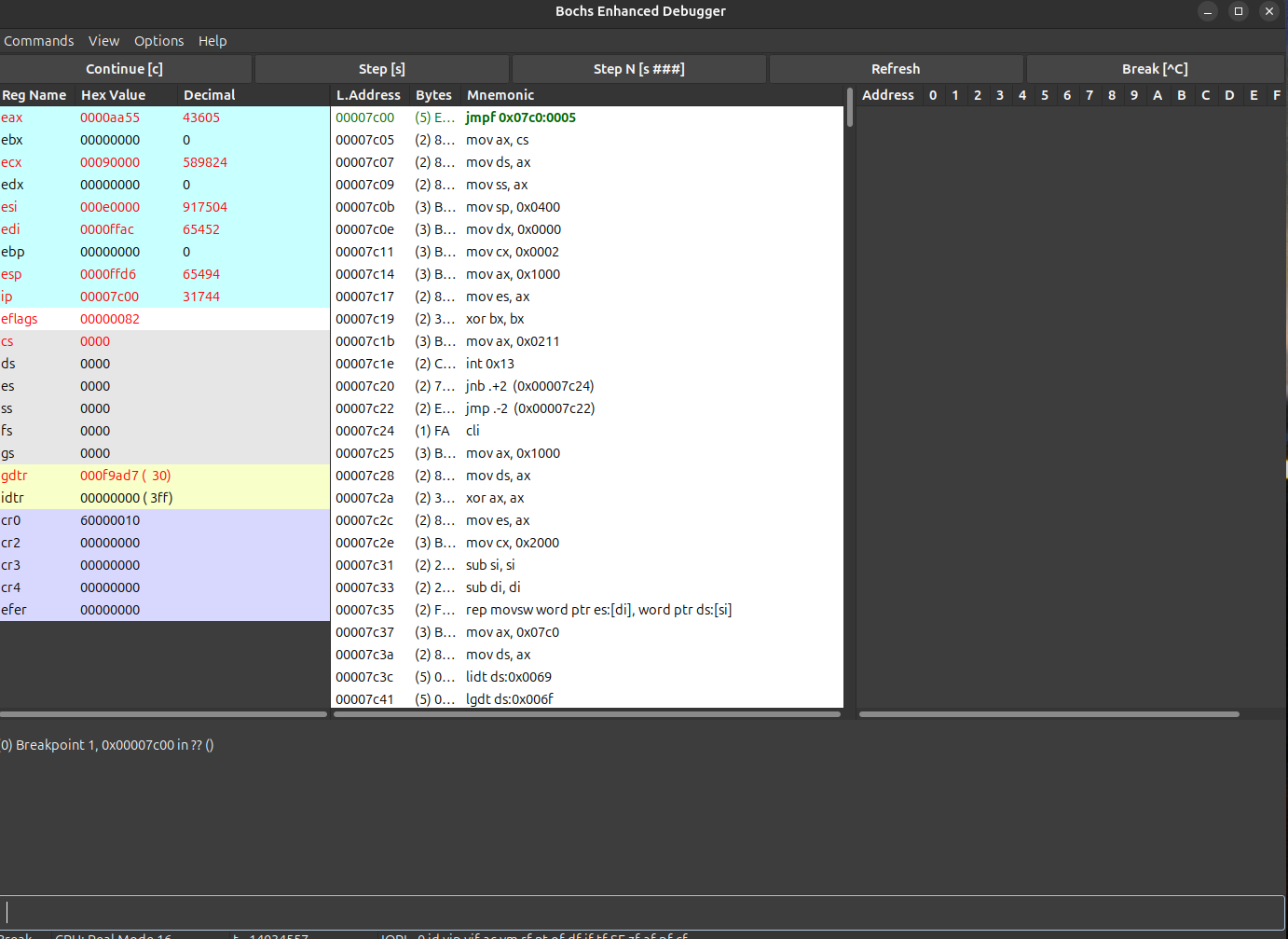
跳转到新代码段：在切换到保护模式后，执行 far jump 指令以跳转到新的代码段。通常，会跳转到新的代码段以开始执行保护模式下的代码。

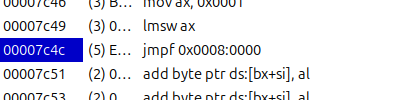
设置栈指针：在保护模式下，通常需要重新设置栈指针（SP）以指向新的栈段。这是因为在实模式下，栈通常在段0x0000下，而在保护模式下通常会有不同的栈段。

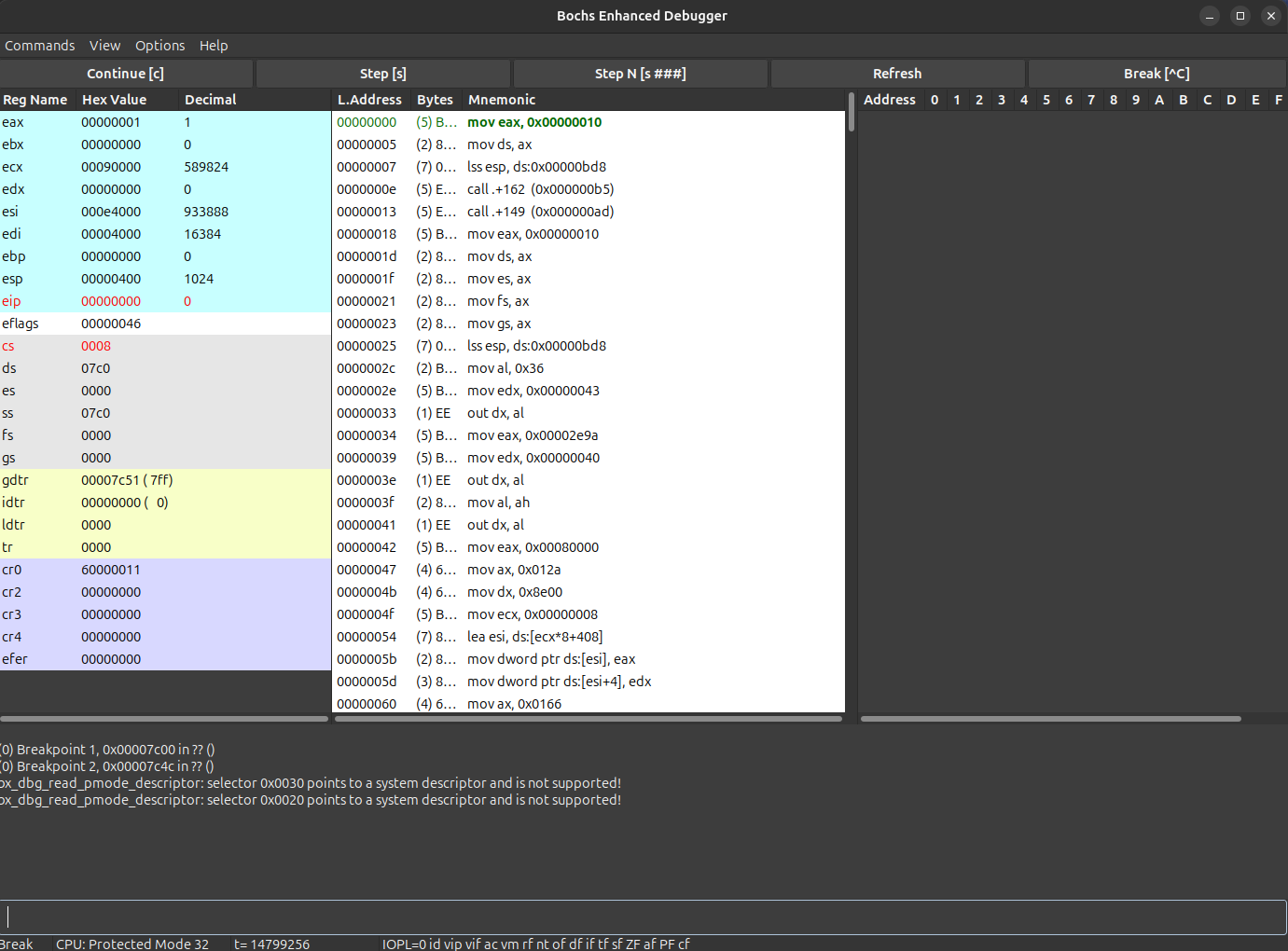
编写保护模式代码：一旦切换到保护模式，需要编写适用于该模式的代码。

·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：

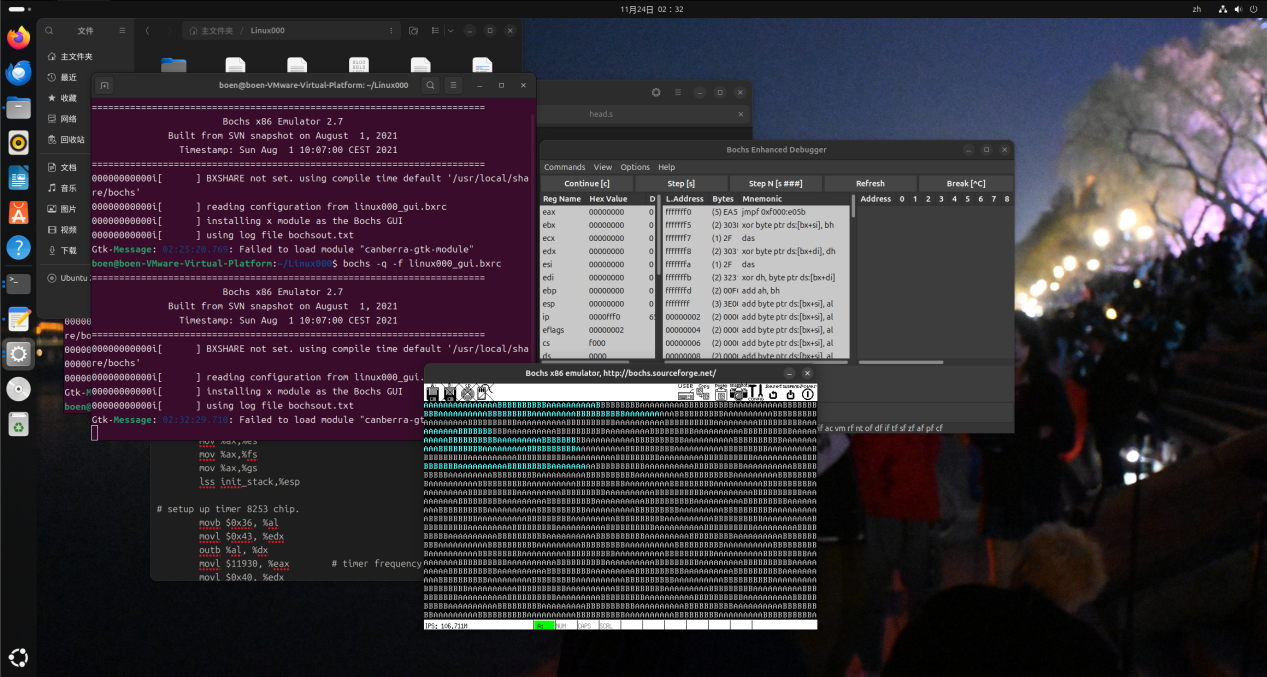
初始化 GDT/IDT

设置系统定时器芯片 8253

初始化 TSS

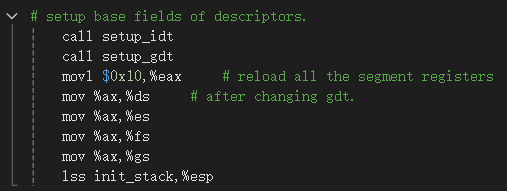
跳转到task0 的用户态程序

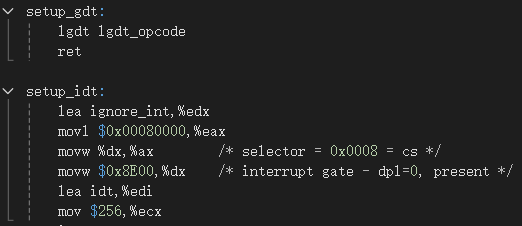
task0 或 task1 的用户态程序在运行时，通过系统调用 int 0x80 向屏幕上打印字符A 或B；时钟中断发生时，内核的中断处理程序实现task0 和task1 的任务切换，运行截图：



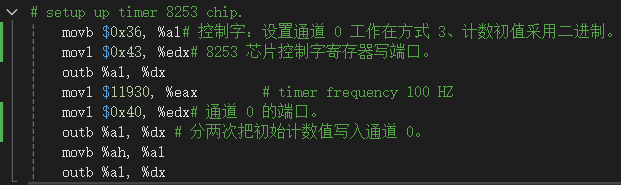
初始化 GDT/IDT

        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）：

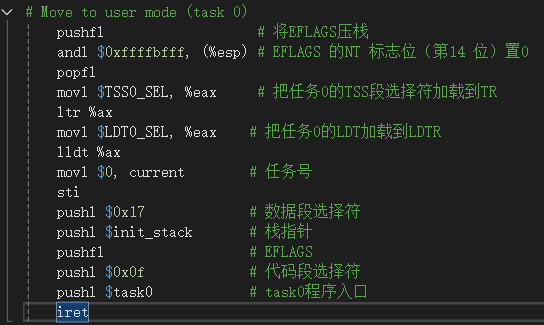




设置系统定时器芯片：

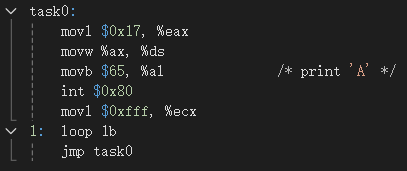


跳转到 task 0：



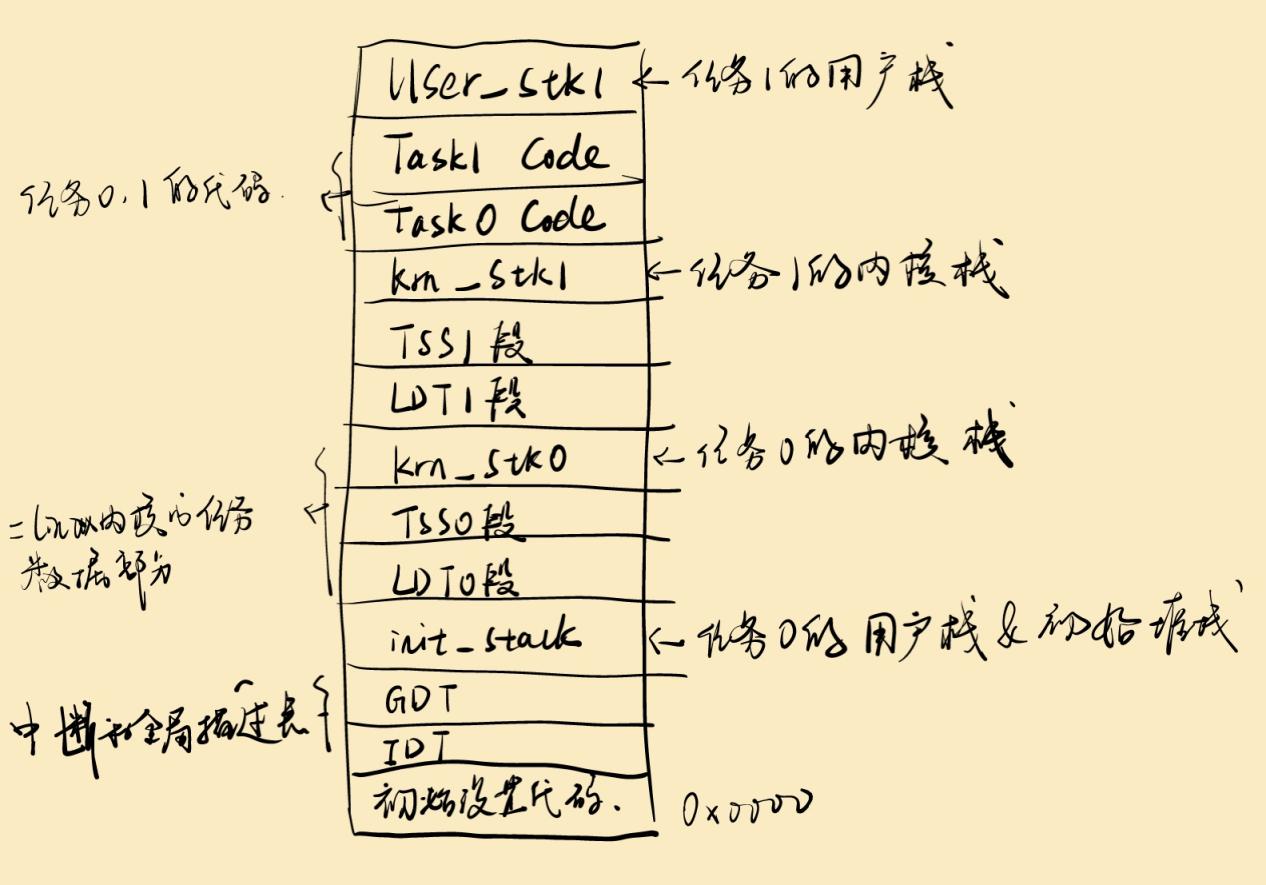
最后执行 iret 时，iret 会把栈顶弹出，更新CS 和 IP，然后把下一个栈顶弹出，更新EFLAGS，然后把下两个栈顶弹出，更新 SS 和ESP，此时便跳转到 task0 下执行。

task0 代码：



task0 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 |

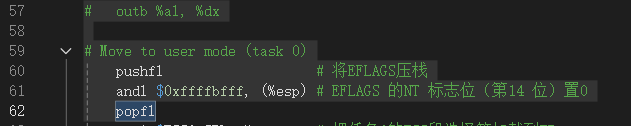
1. 数据段

| 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 |
| ldt0 | 0xbe0 | 0xbf7 |
| tss0 | 0xbf8 | 0xc5f |
| ldt1 | 0xc60 | 0xe77 |
| tss1 | 0xe78 | 0xedf |

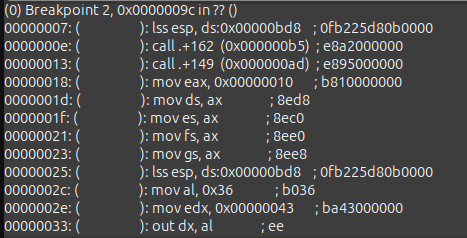
1. 代码段

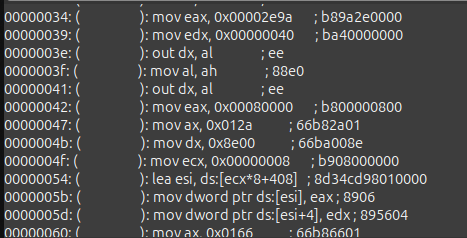
| 名称 | 起始地址 | 终止地址 |
| --- | --- | --- |
| startup\_32 | 0x00 | 0xac |
| setup\_gdt | 0xad | 0xb4 |
| setup\_idt | 0xb5 | 0xe4 |
| write\_char | 0xe5 | 0x113 |
| ignore\_int | 0x114 | 0x129 |
| timer\_interrupt | 0x12b | 0x165 |
| system\_interrupt | 0x166 | 0x17c |
| task0 | 0x10e0 | 0x10f3 |
| task1 | 0x10f4 | 0x1107 |

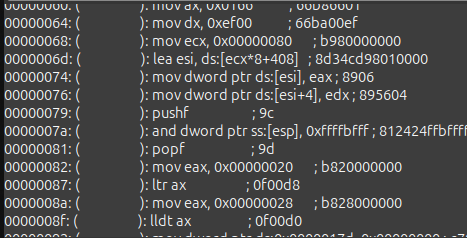
·简述 head.s 57 至 62 行在做什么？

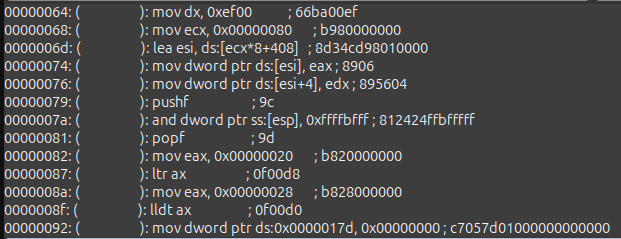
用vs打开偷瞄一眼，很明显不是需要的字段，于是开始调试：  
 

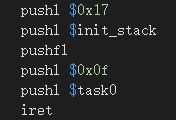
首先需要找到57-62行的位置，所以先在第56行设置断点，然后对57-62行进行反汇编（即disasm 56address 7）出现这些东西（为什么命令区不能放大？）：









同样的，在源文件中找到这些行的内容：  
 

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。

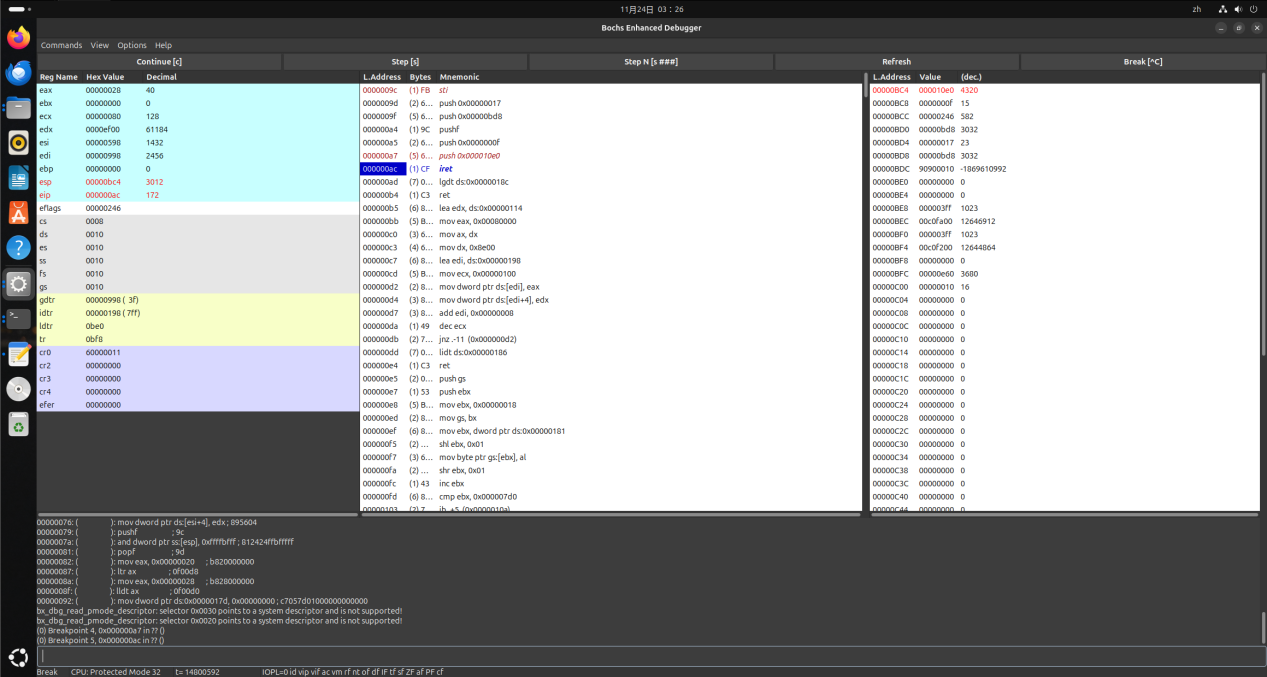
恢复SS（堆栈段寄存器）状态： IRET指令会从堆栈中弹出保存的SS值（中断发生前的堆栈段选择子），并将该值存入SS寄存器。

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？

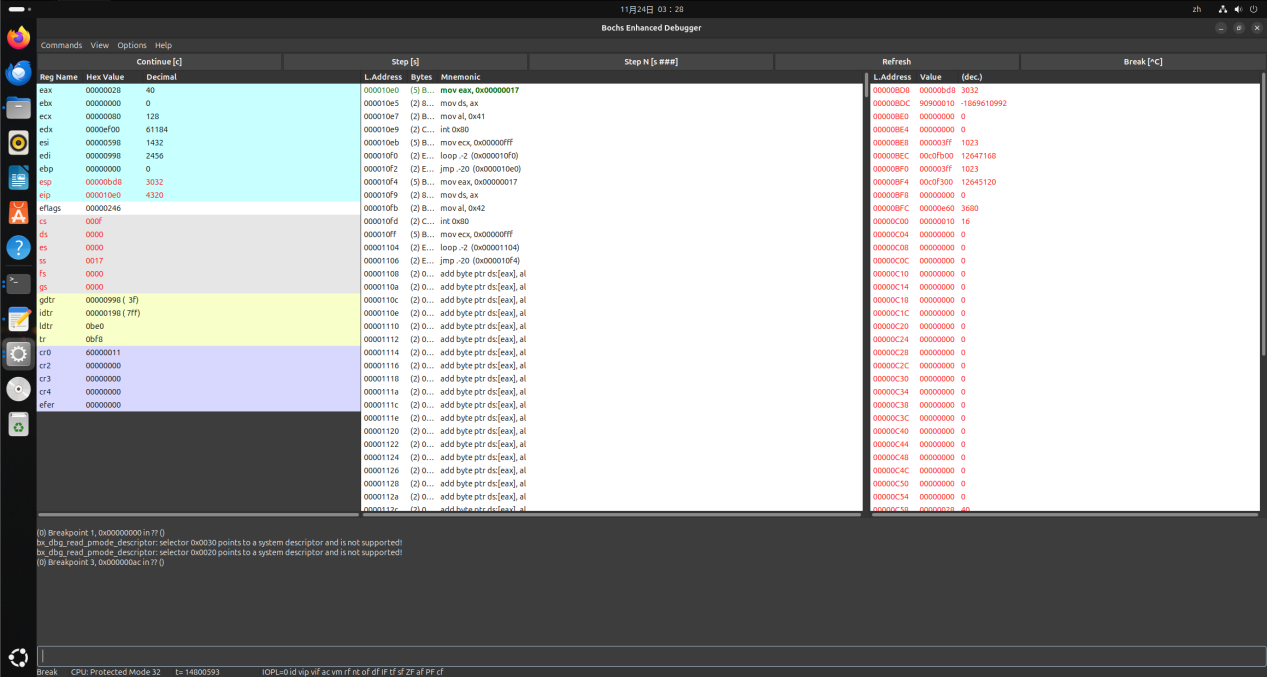
在iret位置设置断点，并且continue检查前后：

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。

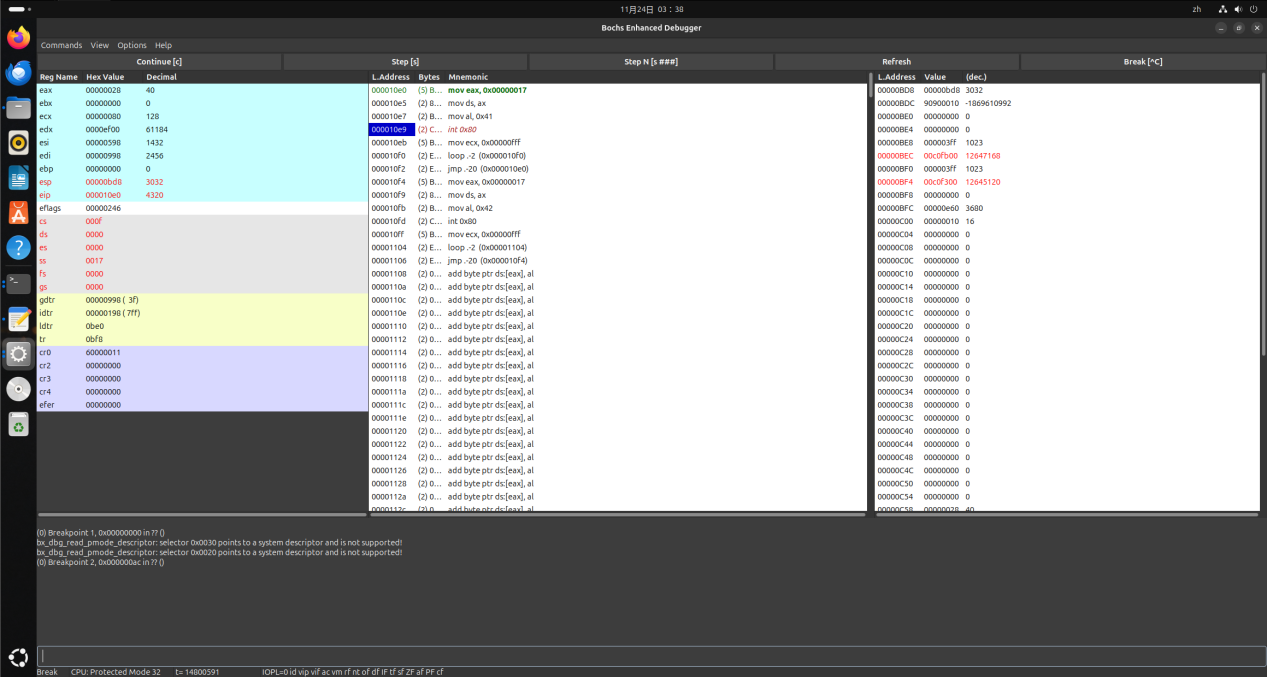
接着，它弹出标志寄存器（EFLAGS）的值，以恢复标志状态。

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。

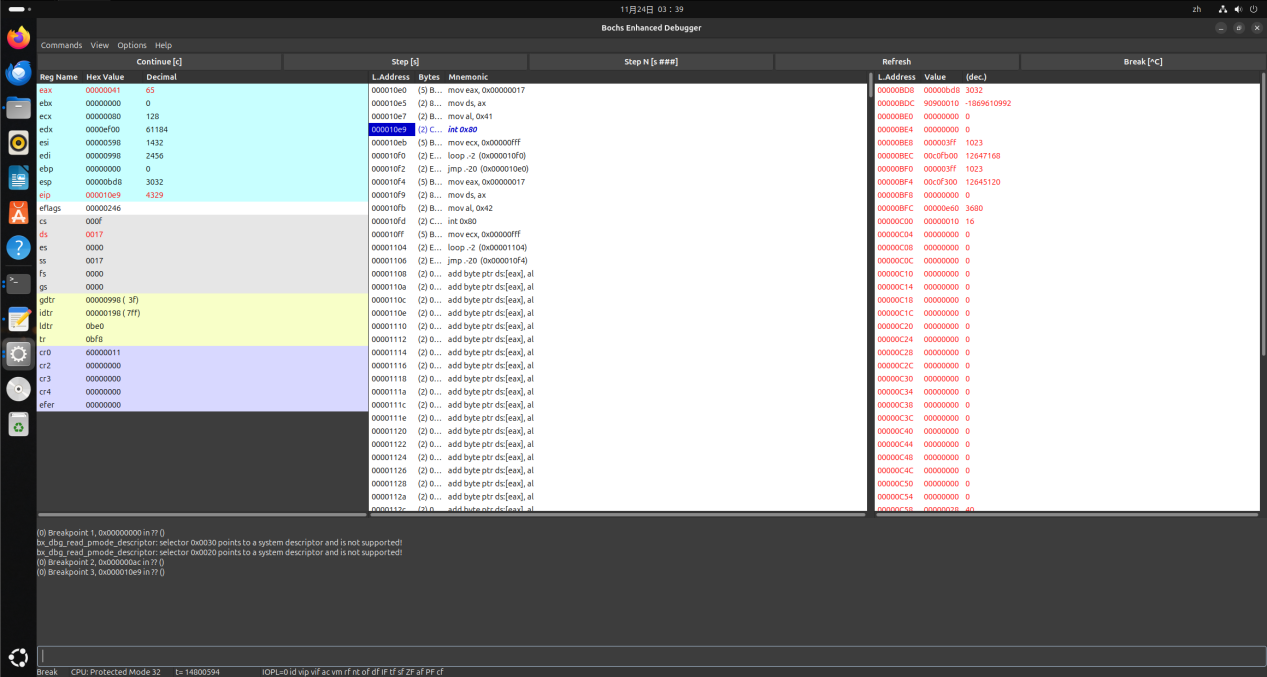
iret 将控制传递到任务 0 中的下一条指令，以继续程序的正常执行。

·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：

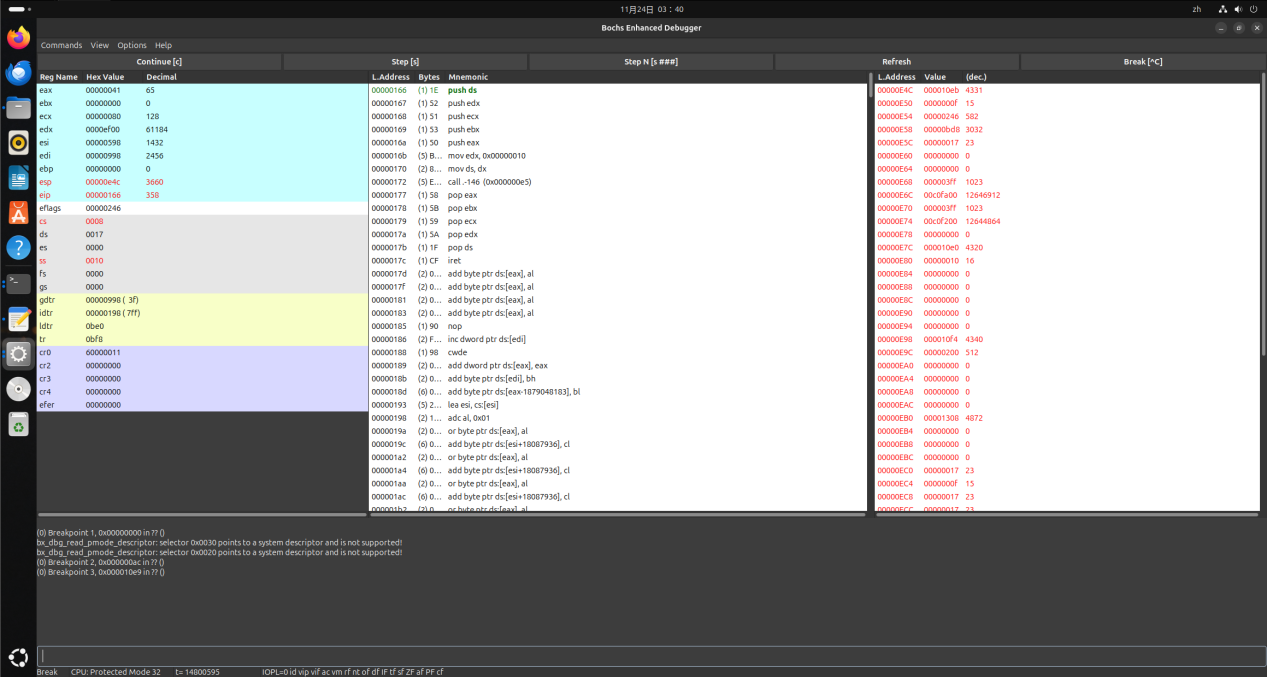


当任务 0 希望执行系统调用时，它会将系统调用号和相关参数加载到寄存器中。

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%