卡通人物

描述已自动生成

**操作系统**

**课程实验报告**

|  |  |  |  |
| --- | --- | --- | --- |
| 实验名称 | 调试分析 Linux 0.00 引导程序 | | |
| 姓名 | 常添 | 院系 | 计算学部 |
| 班级 | 2203102 | 学号 | 2022111699 |
| 任课教师 | 郑铁然 | 指导教师 | 郑铁然 |
| 实验地点 | G709 | 实验时间 | 2024.11.17 |

****

**实验1：调试分析 Linux 0.00 引导程序**

1. **实验环境的配置**

**0.1 bochs的安装和配置**

STEP 1：首先下载并解压bochs-2.7.tar.gz压缩包，运行:

tar -xvzf bochs-2.7.tar.gz

将压缩包解压至虚拟机桌面文件夹。

社交网络的手机截图

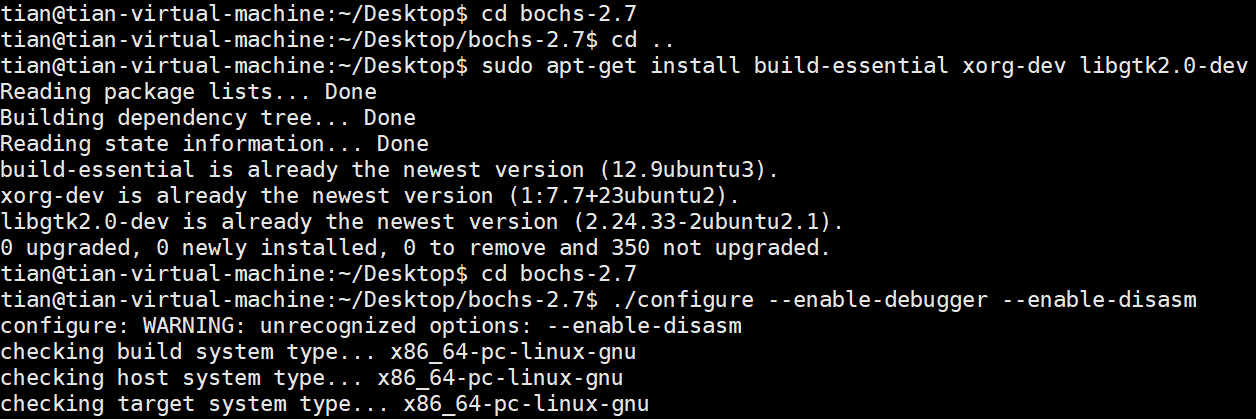
描述已自动生成图0.1.1下载并解压bochs-2.7.tar.gz压缩包

STEP 2：进入bochs-2.7安装目录，并运行：

sudo apt update

sudo apt-get install build-essential xorg-dev libgtk2.0-dev

以获取必要的包和库

图0.1.2获取必要的包和库

STEP 3：在bochs-2.7安装目录下，运行配置，然后编译并运行：

cd ~/Desktop/bochs-2.7

./configure --enable-debugger --enable-disasm

make -j 2

sudo make install

文本

描述已自动生成图0.1.3编译并安装bochs-2.7

STEP 4：进一步安装bochs工具：

sudo apt-get install bochs bochs-sdl bochs-x

**文本

描述已自动生成0.2 安装并首次启动Linux0.00引导程序**

STEP 1：安装git工具，并在桌面克隆实验仓库，进入实验目录文件夹：

sudo apt install git

git clone <https://gitee.com/guojunos/Linux000.git>

cd Linux000/Linux000\ code

文本

描述已自动生成图0.2.1克隆并进入实验目录

STEP 2：安装必要的编译工具：

sudo apt-get update

sudo apt-get install bin86

图0.2.2安装必要工具

STEP 3：执行编译Linux0.00的脚本：

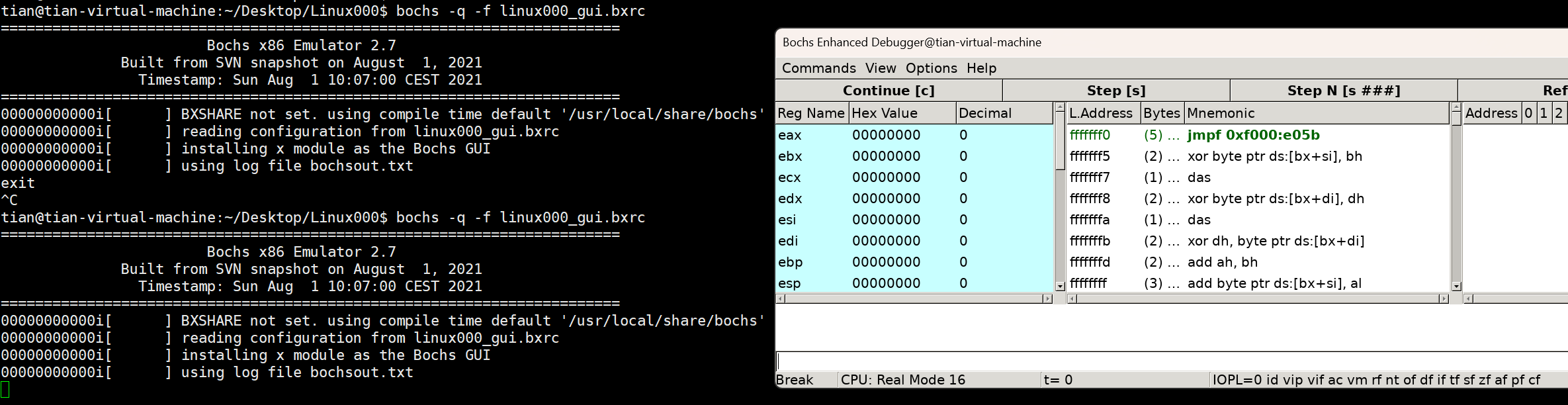
make all

文本

描述已自动生成图0.2.3执行编译Linux0.00的脚本

STEP 4：在Linux000目录下，调用linux000\_gui.bxrc配置文件（注释掉Windows相关的部分，对Linux相关部分解除注释），并启动bochs：

bochs -q -f linux000\_gui.bxrc

图0.2.4调用linux000\_gui.bxrc配置文件并启动bochs

STEP 5：在Bochs Enhanced Debugger窗口的命令行输入c，然后回车运行。此时在仿真窗口Bochs x86 emulator上会展示程序的运行过程。

(Bochs Enhanced Debugger) c

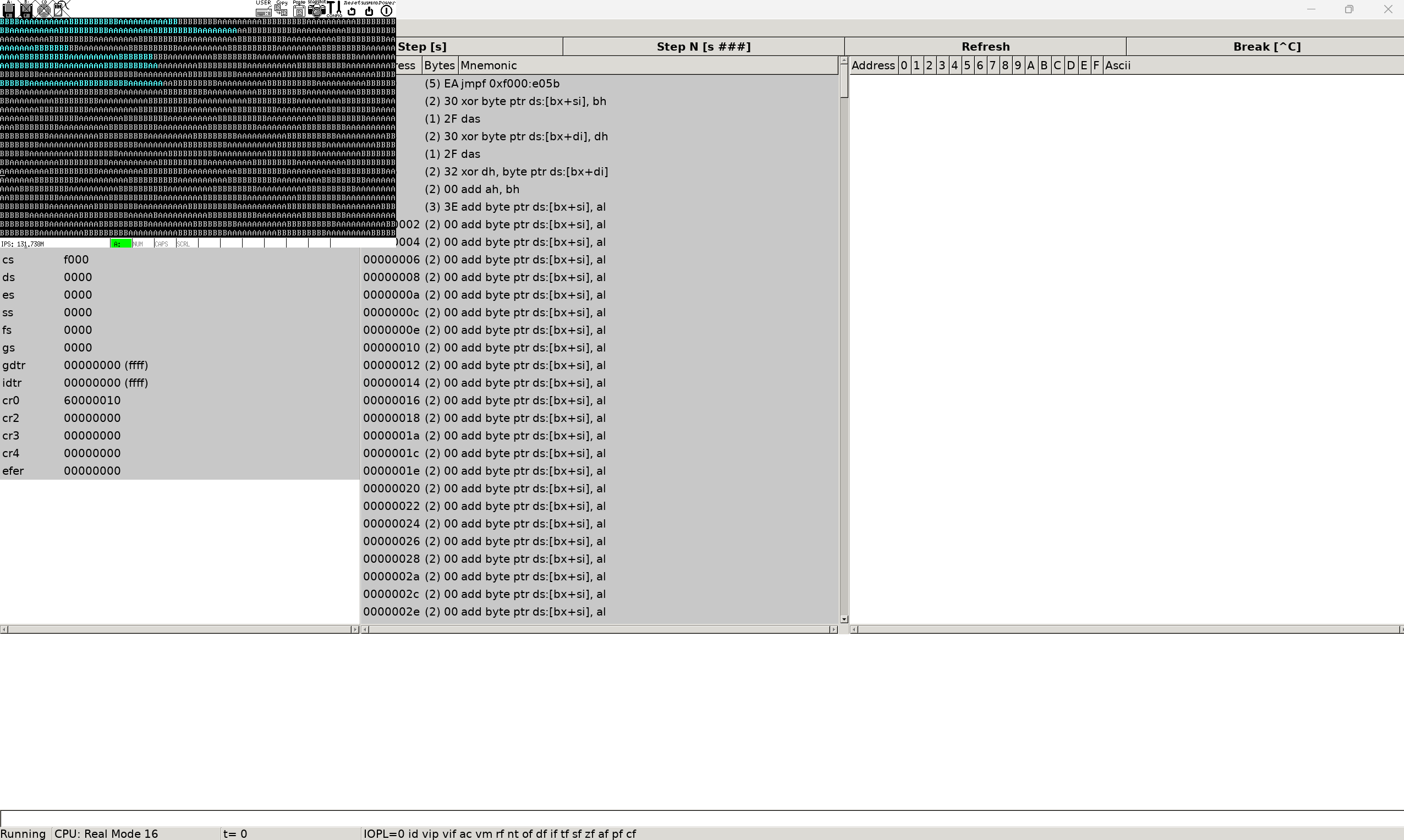


图0.2.5 Bochs Enhanced Debugger调试界面与Bochs x86 emulator运行窗口的展示

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. **实验任务记录（问题回答）**
   1. **请简述 head.s 的工作原理：**

**2.1.1 概述**

* head.s 中包含了32位模式下的启动代码，用来初始化全局描述符表 (GDT)、中断描述符表 (IDT) 以及任务状态段 (TSS)，以支持简单的多任务调度。
* 代码中有两个 L3 任务，用于演示基本的任务切换机制，内核代码位于内存地址 0x10000。

**2.1.2 启动代码 (startup\_32)**

* 加载段寄存器：
  + 初始化 ds（数据段选择子）并设置栈指针 esp 到 init\_stack。
* 调用 setup\_idt 和 setup\_gdt：
  + setup\_idt 用于初始化 IDT 表，设置默认和系统调用的中断处理程序。
  + setup\_gdt 用于加载 GDT 表，设置不同的段描述符（包括代码段、数据段和任务段）。
* 重载段寄存器：
  + 确保在 GDT 更新后重新加载 ds、es、fs 和 gs。
* 配置 8253 定时器：
  + 设定时钟频率为 100 Hz，配置时钟中断以支持任务调度。
* 设置中断描述符：
  + 为定时器中断和系统调用设置中断描述符。

**2.1.3 多任务支持**

* TSS 和任务切换：
  + 初始化 TSS（任务状态段），将 TSS 加载到任务寄存器 (ltr)，并设置 LDT（本地描述符表）。
  + 使用 ljmp 指令进行任务间切换。

**2.1.4 中断处理程序**

* 默认中断处理 (ignore\_int)：
  + 打印字符 'C' 以指示中断发生，并返回。
* 时钟中断处理 (timer\_interrupt)：
  + 定时器中断用于实现多任务切换。检查当前任务并跳转到对应的 TSS。
* 系统调用处理 (system\_interrupt)：
  + 打印字符并返回，是一个简单的系统调用处理演示。

**2.1.5 输出字符功能 (write\_char)**

* 显示字符：
  + 将字符显示到屏幕上的显存区域，通过设置显存选择子和操作寄存器进行字符输出。

**2.1.6 数据结构**

* GDT：
  + 包含 NULL 描述符、代码段、数据段、显存段、TSS 段和 LDT 段。
* TSS：
  + 定义两个任务 (tss0 和 tss1) 的任务状态段结构，包含任务切换所需的寄存器状态和栈指针。
* LDT：
  + 每个任务都有独立的 LDT，定义任务私有的段选择子。

**2.1.7 任务实现 (task0 和 task1)**

* task0 和 task1 是两个简单的任务，它们分别打印字符 'A' 和 'B' 并进入一个循环。通过定时器中断和任务切换机制，它们实现了基本的多任务演示。

**2.2 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：**

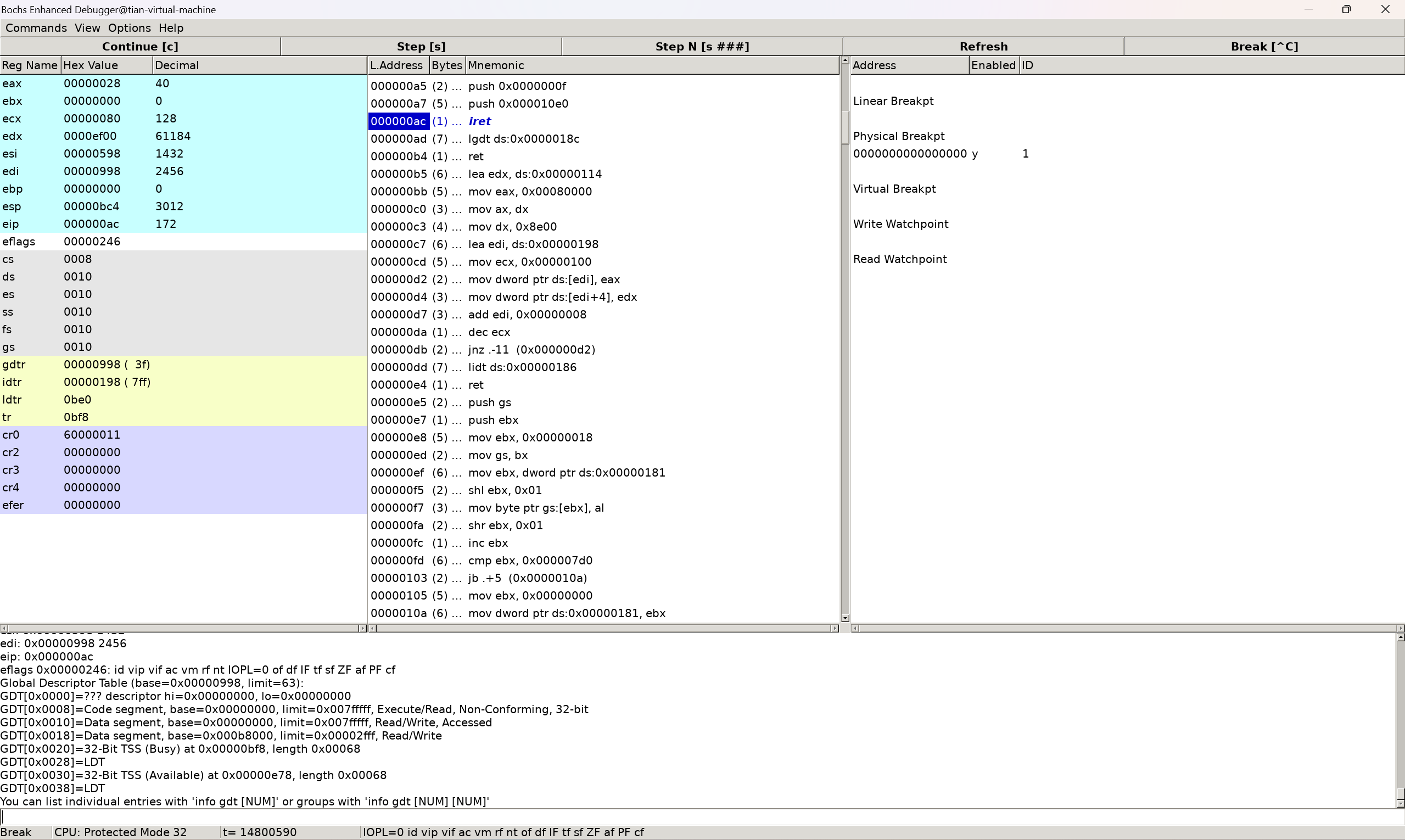


图2.2.1 Bochs Enhanced Debugger调试界面中查看内存分布情况

**2.2.1在Bochs Enhanced Debugger中运行：**

(Bochs Enhanced Debugger) b 0x00

(Bochs Enhanced Debugger) c

(Bochs Enhanced Debugger) info gdt

从截图获得信息如下：

Global Descriptor Table (base=0x00000998, limit=63):

GDT[0x0000]=??? descriptor hi=0x00000000, lo=0x00000000

GDT[0x0008]=Code segment, base=0x00000000, limit=0x007fffff, Execute/Read, Non-Conforming, 32-bit

GDT[0x0010]=Data segment, base=0x00000000, limit=0x007fffff, Read/Write, Accessed

GDT[0x0018]=Data segment, base=0x000b8000, limit=0x00002fff, Read/Write

GDT[0x0020]=32-Bit TSS (Busy) at 0x00000bf8, length 0x00068

GDT[0x0028]=LDT

GDT[0x0030]=32-Bit TSS (Available) at 0x00000e78, length 0x00068

GDT[0x0038]=LDT

**2.2.2 对内存分布状况的总结如下：**

**每个段的基址和限长：**

* **GDT 基址**：0x00000998
* **GDT 限长**：63 字节

**各段的具体信息：**

* **GDT[0x0008]** (代码段)：
  + **基址**：0x00000000
  + **限长**：0x007FFFFF（8 MB）
  + **属性**：32 位、可执行、可读
* **GDT[0x0010]** (数据段)：
  + **基址**：0x00000000
  + **限长**：0x007FFFFF（8 MB）
  + **属性**：32 位、可读/写
* **GDT[0x0018]** (显存数据段)：
  + **基址**：0x000B8000（显存地址）
  + **限长**：0x00002FFF（12 KB）
  + **属性**：32 位、可读/写
* **GDT[0x0020]** (任务状态段 TSS)：
  + **基址**：0x00000BF8
  + **长度**：0x00068（104 字节）
  + **属性**：32 位 TSS（忙状态）
* **GDT[0x0030]** (任务状态段 TSS)：
  + **基址**：0x00000E78
  + **长度**：0x00068（104 字节）
  + **属性**：32 位 TSS（可用状态）

**内存分布记录:**

* **代码段 (CS)**：
  + **起始地址**：0x00000000
  + **终止地址**：0x007FFFFF（8 MB）
  + **属性**：32 位、可执行、可读
* **数据段 (DS)**：
  + **起始地址**：0x00000000
  + **终止地址**：0x007FFFFF（8 MB）
  + **属性**：32 位、可读/写
* **显存数据段 (GDT[0x0018])**：
  + **起始地址**：0x000B8000
  + **终止地址**：0x000B8000 + 0x2FFF = 0x000BAFFF
  + **属性**：32 位、可读/写
* **任务状态段 (TSS)**：
  + **第一个 TSS 段**：
    - **基址**：0x00000BF8
    - **长度**：0x00068（104 字节）
  + **第二个 TSS 段**：
    - **基址**：0x00000E78
    - **长度**：0x00068（104 字节）

**栈段分析:**

栈段的基址由 ss 寄存器和 GDT 中的定义确定。查看 esp 寄存器来了解栈的当前位置。

* **栈指针 (esp)**：当前值为 0x00000BC4。
* **栈段 (ss 寄存器选择子应为 0x0010 数据段)**：
  + **起始地址**：0x00000000
  + **终止地址**：0x007FFFFF（8 MB）

**2.3 简述 head.s 57 至 62 行在做什么？**

head.s 中的第 57 至 62 行执行了准备从内核模式切换到用户模式并启动第一个用户任务的过程。这几行指令的作用如下：

* pushl $0x17：将段选择子 0x17 压入栈。0x17 是用户模式的数据段选择子，对应 GDT 中一个具有用户权限的段。
* pushl $init\_stack：将 init\_stack 的地址压入栈，这将作为任务的栈指针 (ESP)，指示用户模式任务将使用的栈地址。
* pushfl：将当前 EFLAGS 寄存器的值压入栈，保存当前的标志位状态。
* pushl $0x0f：将用户模式代码段选择子 0x0f 压入栈。此选择子对应 GDT 中一个具有用户权限的代码段。
* pushl $task0：将 task0 的地址（用户任务的起始地址）压入栈，用于设置任务的指令指针 (EIP)。
* iret：执行中断返回指令，从栈中依次弹出 EIP、CS、EFLAGS、ESP 和 SS。这将切换到用户模式，并从 task0 的地址开始执行。

目的：

这几行代码的主要目的是从内核模式切换到用户模式，并启动task0任务。通过设置段选择子、栈指针和指令指针，然后使用 iret 来恢复上下文，完成对用户模式的切换和新任务的启动。这是实现多任务操作系统的重要步骤，使系统能够在不同特权级之间进行上下文切换。

* 1. **简述 iret 执行后， pc 如何找到下一条指令？**

iret 指令的主要功能是从中断或任务切换中返回，并恢复被中断时保存的寄存器状态。这包括程序计数器 (PC) 的恢复，使 CPU 能够继续执行被中断前的代码或启动新的任务。

iret 执行的过程：

iret 指令会按照以下顺序从栈中弹出寄存器的值并将其恢复：

1. 恢复 EIP（程序计数器）： iret 从栈中弹出地址并加载到 EIP 寄存器中，这个地址是下一条要执行的指令的地址。
2. 恢复 CS（代码段寄存器）： iret 接着从栈中弹出代码段选择子，并将其加载到 CS 寄存器中。CS:EIP 组合指定了完整的代码段内存地址，指向下一条要执行的指令。
3. 恢复 EFLAGS（标志寄存器）： iret 从栈中弹出 EFLAGS 寄存器的值，恢复中断前的标志状态。
4. 在从内核模式切换到用户模式的情况下： 如果 iret 是从更高特权级（如内核模式）切换到更低特权级（如用户模式），它还会从栈中恢复 ESP（栈指针）和 SS（栈段寄存器），用于设置用户模式的栈。

PC 如何找到下一条指令：

* EIP 的作用：EIP 是程序计数器寄存器，存储了 CPU 当前要执行的指令的地址。当 iret 执行完毕并从栈中恢复了 EIP，CPU 就会继续执行 CS:EIP 指向的那条指令。
* 地址的组合：CS 寄存器提供了段的基址，而 EIP 提供了段内的偏移地址。CS:EIP 组合指向了内存中的一个具体地址，这就是 CPU 将执行的下一条指令的位置。

**2.5 记录 iret 执行前后，栈是如何变化的？**

**2.5.1 iret 执行前的调试和记录**

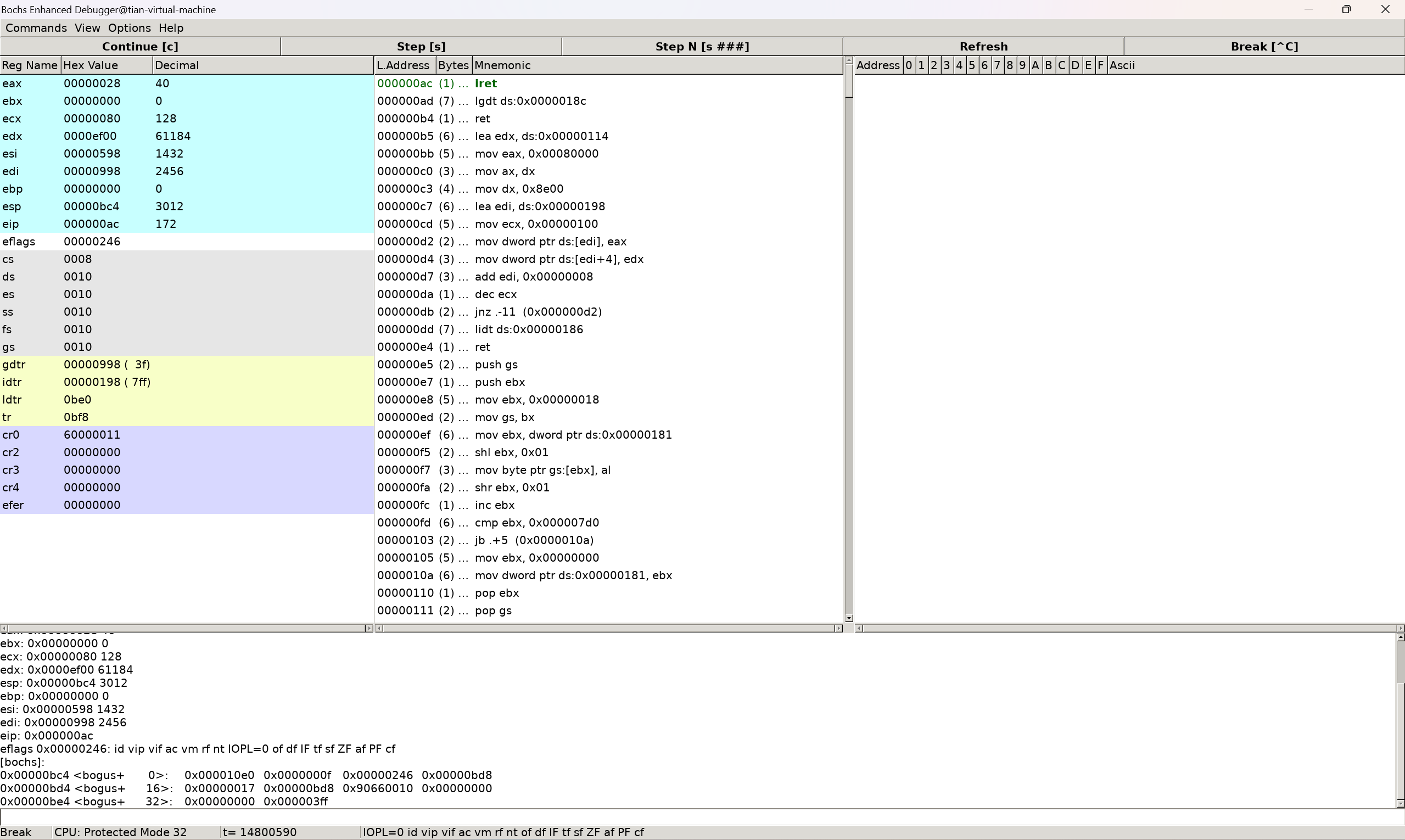
在iret处添加断点，然后进行如下调试过程，并打印内存和寄存器信息：

(Bochs Enhanced Debugger) b 0xac

(Bochs Enhanced Debugger) c

(Bochs Enhanced Debugger) r

(Bochs Enhanced Debugger) x/10x esp

****图2.5.1 Bochs Enhanced Debugger调试界面中iret 执行前的内存和寄存器情况

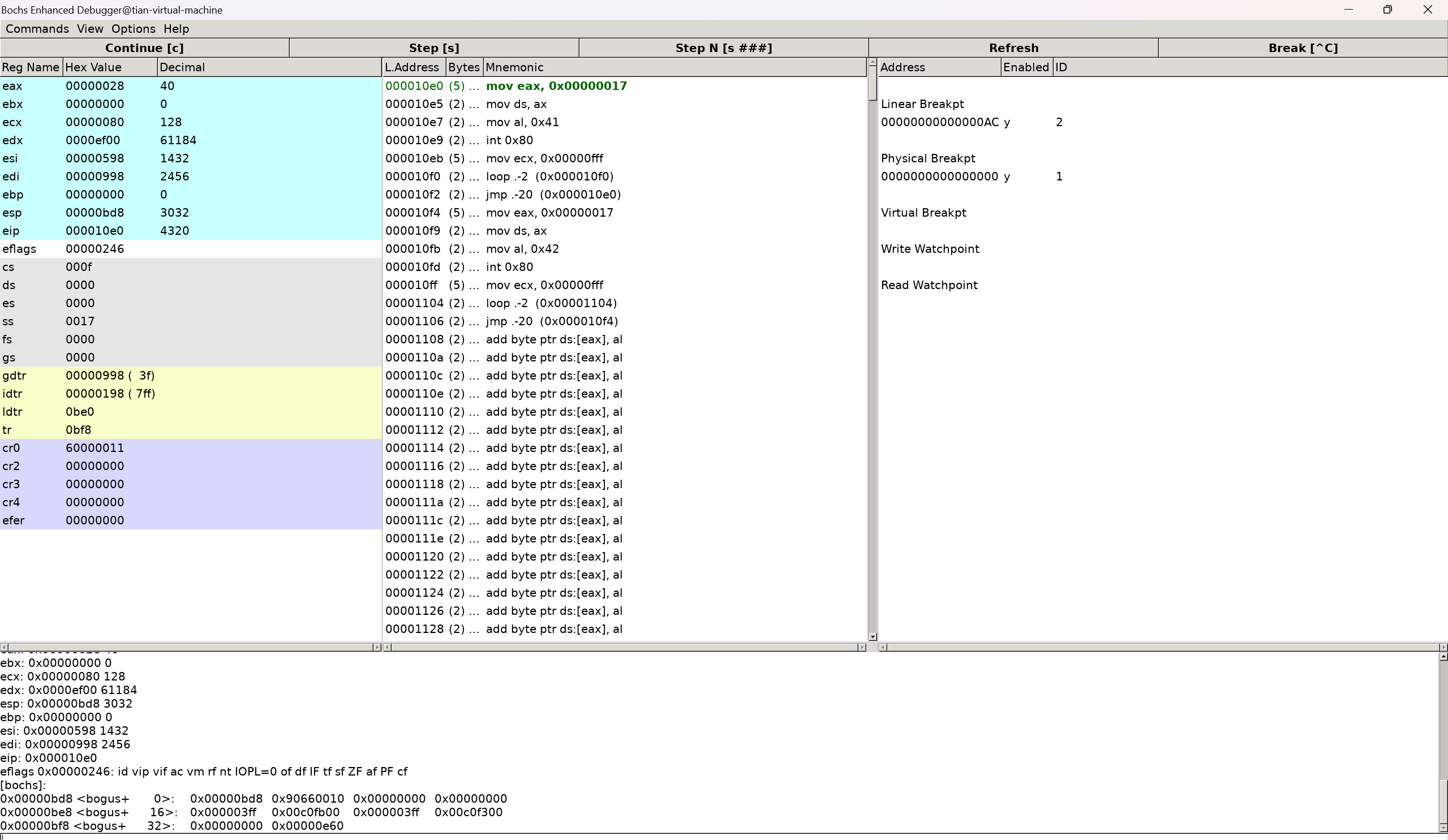
**2.5.2 iret 执行后的调试和记录**

接下来单步执行iret，并打印内存和寄存器信息：

(Bochs Enhanced Debugger) s

(Bochs Enhanced Debugger) r

(Bochs Enhanced Debugger) x/10x esp

****图2.5.2 Bochs Enhanced Debugger调试界面中iret 执行后的内存和寄存器情况

**2.5.3 分析iret 执行前后栈的变化：**

iret 执行前，栈中保存了返回地址 EIP、代码段选择子 CS 和标志寄存器 EFLAGS，以及可能涉及到的特权级切换时的 SS 和 ESP。iret 执行后，这些值被弹出，esp 上移，指向下一个栈顶位置，而程序计数器和段寄存器也更新以继续执行恢复的程序位置。下面是结合图片的详细分析。

**iret 执行前的栈内容**

在 iret 执行之前，esp 指向 0x00000bc4，栈上的内容如下：

0x00000bc4: 0x000010e0 ; EIP - 下一条指令的地址

0x00000bc8: 0x0000000f ; CS - 代码段选择子

0x00000bcc: 0x00000246 ; EFLAGS - 标志寄存器

0x00000bd0: 0x00000bd8 ; （如果涉及到特权级切换，还会保存用户模式的 ESP）

0x00000bd4: 0x00000017 ; （如果涉及到特权级切换，还会保存用户模式的 SS）

**iret 执行后的栈和寄存器状态**

在 iret 执行之后，栈指针 esp 指向 0x00000bd8，并且 EIP 和 CS 已被更新：

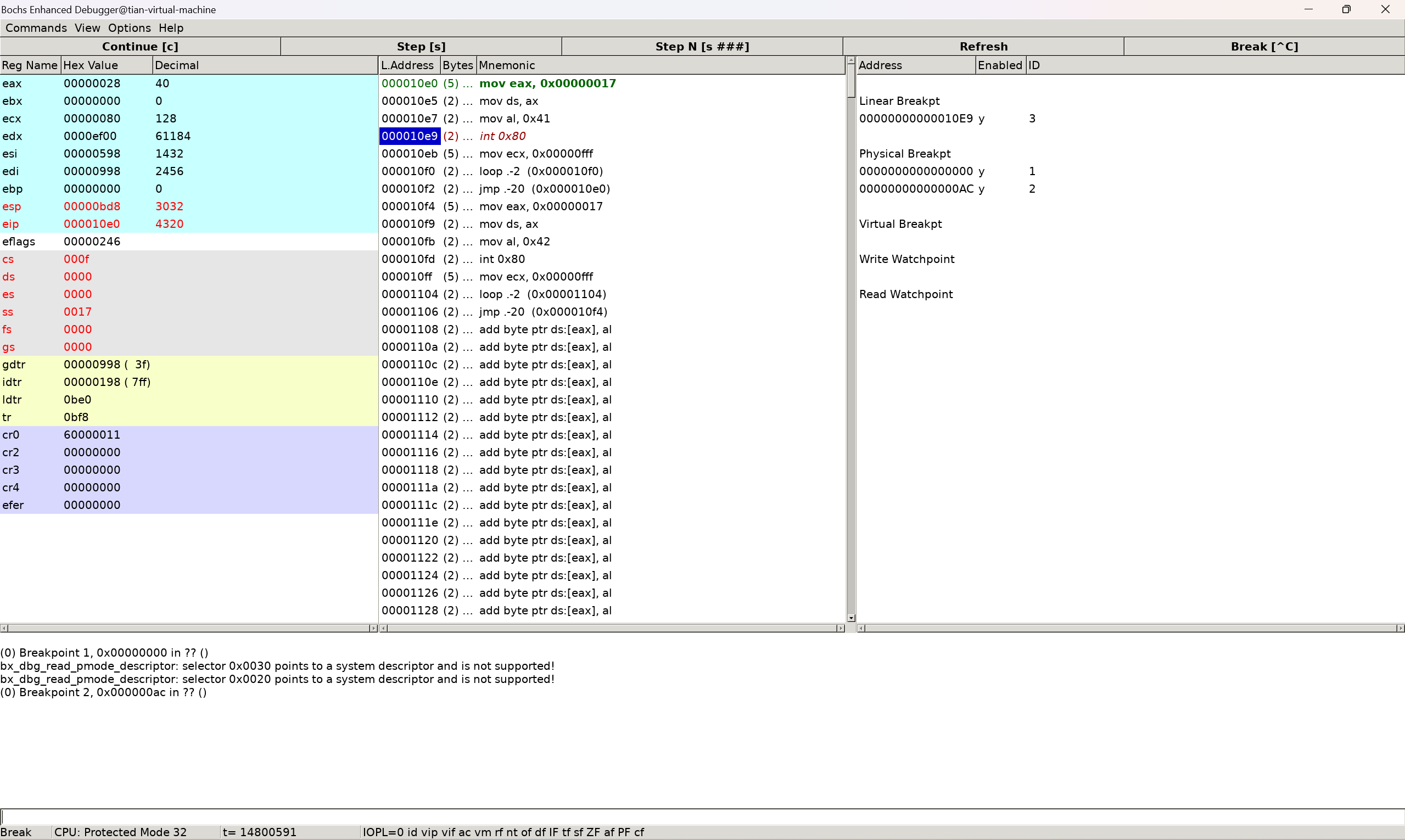
* **新寄存器状态**：
  + eip: 0x000010e0（从栈中弹出并设置，指向下一条执行指令的地址）
  + cs: 0x0000000f（代码段选择子从栈中弹出并设置）
  + eflags: 0x00000246（从栈中弹出并恢复）

**2.5.4 栈变化的解释：**

* **iret 指令从栈中弹出 EIP、CS、EFLAGS**：
  + 将 EIP 设置为 0x000010e0，这将指向下一条需要执行的指令。
  + 将 CS 设置为 0x0000000f，确定代码段选择子。
  + 将 EFLAGS 恢复为 0x00000246，确保标志寄存器的状态和中断前一致。
* **栈指针 esp 上移**：
  + esp 更新为 0x00000bd8，表示已从栈中弹出 EIP、CS 和 EFLAGS，栈内容已被消费。
  1. **当任务进行系统调用时，即 int 0x80 时，记录栈的变化情况。**

**2.6.1 int 0x80系统调用执行前的调试和记录**

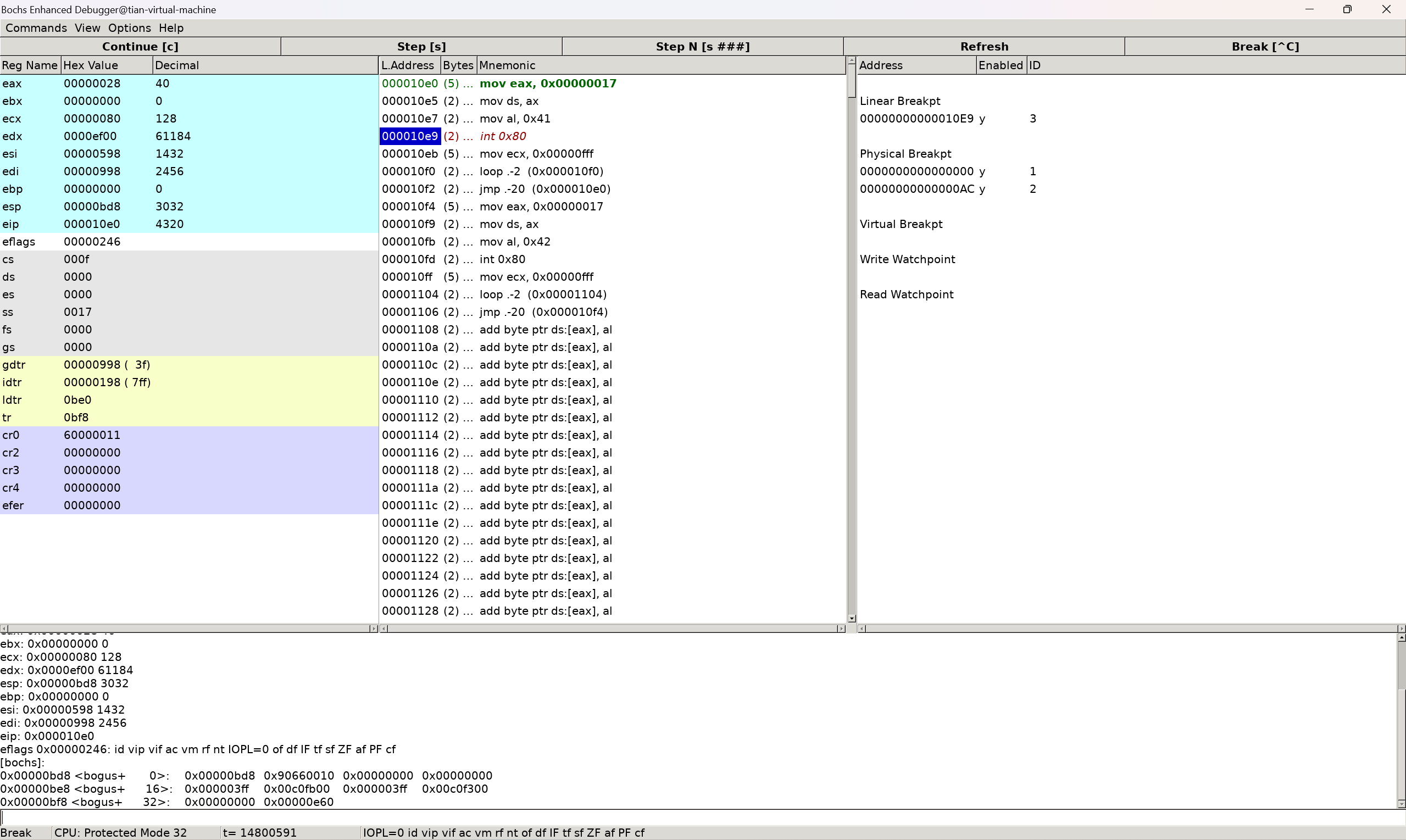
找到int 0x80的位置为0x10e9，并在此处设置断点（双击int 0x80所在位置）：

图2.6.1 Bochs Enhanced Debugger调试界面中在int 0x80的位置设置断点

接下来打印系统调用前的内存和寄存器信息：

(Bochs Enhanced Debugger) r

(Bochs Enhanced Debugger) x/10x esp

图2.6.2 Bochs Enhanced Debugger调试界面中在int 0x80系统调用前的内存和寄存器信息

**2.6.2 int 0x80系统调用执行后的调试和记录**

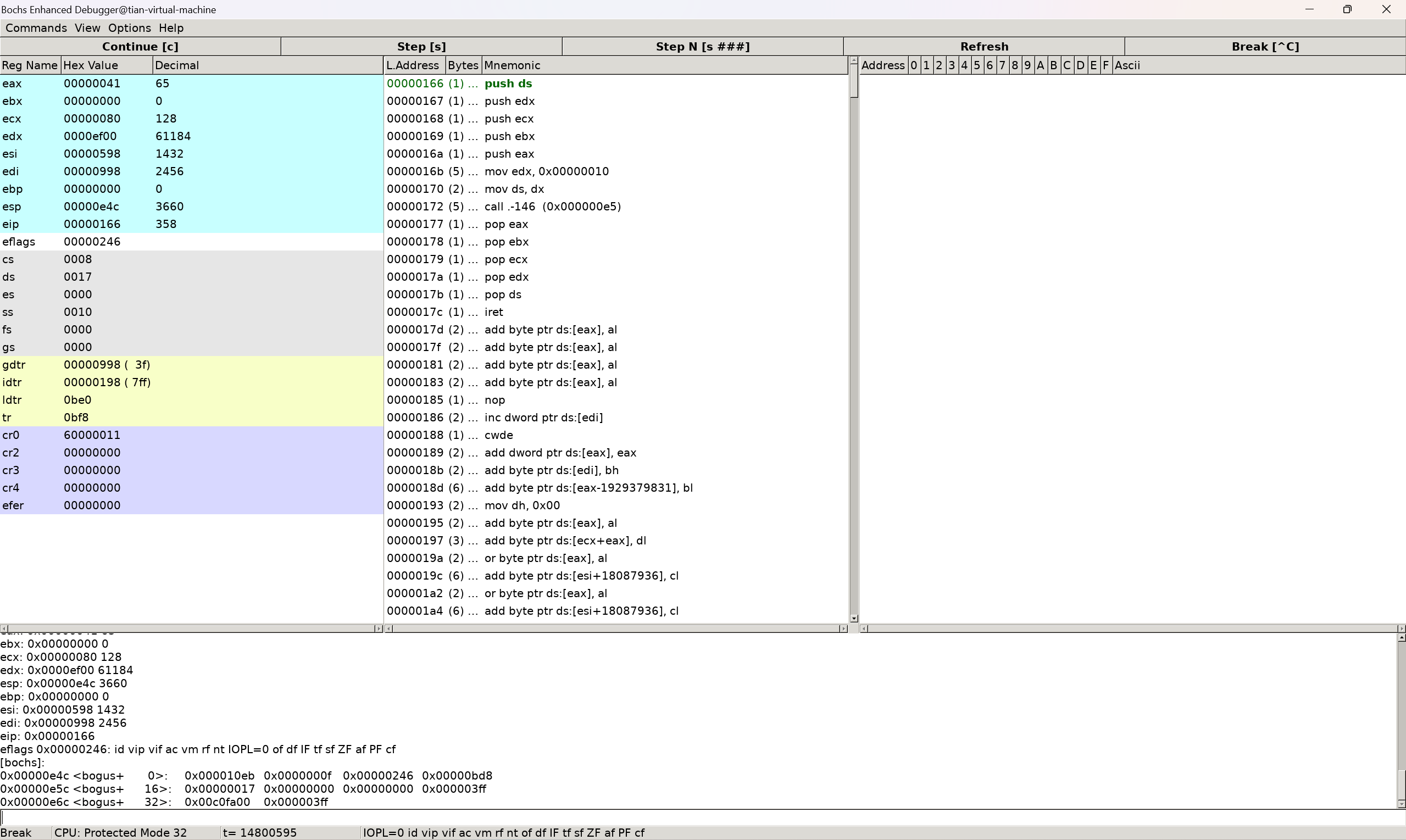
接下来单步执行，并打印系统调用后的内存和寄存器信息：

(Bochs Enhanced Debugger) c

(Bochs Enhanced Debugger) s

(Bochs Enhanced Debugger) r

(Bochs Enhanced Debugger) x/10x esp

图2.6.3 Bochs Enhanced Debugger调试界面中在int 0x80系统调用前的内存和寄存器信息

**2.6.3 int 0x80系统调用执行前后的栈变化分析：**

**1. 调用 int 0x80 前的栈状态**

栈内容（esp 指向的地址 0x00000bd8 开始）：

0x00000bd8: 0x00000bd8 ; 栈顶内容（指示 ESP 的位置）

0x00000bdc: 0x90660010

0x00000be0: 0x00000000

0x00000be4: 0x00000000

此时，栈上保存了一些系统调用参数或调用者状态。

**2. 调用 int 0x80 后的栈状态**

栈内容（esp 指向的地址 0x00000e4c 开始）：

0x00000e4c: 0x000010eb ; 保存返回时的 EIP

0x00000e50: 0x0000000f ; 保存返回时的 CS

0x00000e54: 0x00000246 ; 保存返回时的 EFLAGS

0x00000e58: 0x00000bd8 ; 原始的 ESP

**3. 栈的变化分析**

在调用 int 0x80 系统调用时，CPU 会执行以下步骤：

1. 压入当前的 EIP、CS 和 EFLAGS 到栈中，用于在系统调用结束后恢复程序状态。
2. 转入系统调用处理程序，执行内核代码。
3. 系统调用返回时，iret 指令会从栈中弹出并恢复 EIP、CS 和 EFLAGS，更新栈指针 ESP。

在调用后：

* esp 从 0x00000bd8 更新到 0x00000e4c，表明栈上保存了更多的调用状态。
* 栈上内容已被修改，加入了新的返回地址和标志。

调用前：栈上保存的是调用者的状态，包括参数。调用后：栈上增加了系统调用保存的 EIP、CS 和 EFLAGS，以便在系统调用结束时通过 iret 恢复。

**2.7 调试跟踪 jmpi 0,8 ，解释如何寻址**

jmpi 0, 8 是一条长跳转指令，用于在保护模式下完成段间跳转或代码段切换。它的主要作用是更改代码段寄存器 (CS) 和指令指针 (EIP)，从而跳转到新的代码段的指定地址并继续执行。

具体执行过程如下：

1. 目标段选择子 (8)：
   * 选择子 8 表示目标段，对应全局描述符表（GDT）中的一个代码段描述符。
   * 从 GDT 中获取该段的 基址、限长 和 属性。
2. 偏移地址 (0)：
   * 表示目标段内的偏移量，此处为 0，跳转到目标段的起始位置。
3. 更新寄存器：
   * CS：更新为段选择子 8，代码段基址和属性随之加载到 CS 的隐藏部分。
   * EIP：更新为 0，表示程序计数器指向段的起始位置。
4. 切换到目标段：
   * CPU 开始执行新的段，由 CS:EIP 组合指向的内存地址开始执行代码。