第一章．概述

1.1 80386 保护模式简介

本章的主要内容是介绍 80386 保护模式。

1. 保护模式简介

Intel 推出 x86 架构已近 30 年，刚开始推出的 8086 处理器是一款 16 位的处理器，它标志着 x86 架构的诞生，这种 16 位处理器数据总线是 16 位的，而地址总线是 20 位的，最多可以寻址 1MB 的地址空间。之后的 80286 处理器也是 16 位，但是地址总线有 24 位，而且从 80286 开始 CPU 演变出两种工作模式：实模式和保护模式；而 80386 则是 Intel 推出的 80x86 系列中的第一款 32 位处理器，它的数据总线与地址总线都是 32 位，可以寻址 4G 的地址空间；AMD 公司随后在 2000 年又在 x86 架构的基础上推出了 x86-64 处理器架构，AMD 的处理器可以兼容 32 位的指令集，所以它既是 64 位的又是 32 位的。

在 x86 架构中，16 位的处理器与 32 位处理器所对应的寄存器是有所不同的。像 8086 寄存器组就分为通用寄存器、专用寄存器和段寄存器三类总共 15 个，其中通用寄存器有 AX、BX、CX、DX、SP、BP、DI 及 SI，专用寄存器包括 IP、SP 和 FLAGS 三个 16 位寄存器，而段寄存器则有 CS、DS、SS、ES，这些寄存器都是 16 位的。32 位 x86 架构对应的寄存器则共有 34 个，其中包括 EAX、EBX、ECX、EDX、ESI、EDI、EBP、ESP 8 个 32 位的通用寄存器；6 个 16 位的段寄存器 CS、DS、SS、ES、FS、GS，相比 8086 增加了 FS 和 GS；GDTR、LDTR、IDTR 和 TR 四个系统地址寄存器；EFLAGS、EIP、CR0---CR3 6 个状态和控制寄存器，在这里标志寄存器 EFLAGS 与指令指针寄存器 EIP 都从 16 位进化到了 32 位；还有就是增加了一些调试寄存器、段描述符寄存器以及测试寄存器。

表 1-18086 和 86386 的寄存器的对比

	8086 的寄存器	80386 的寄存器
通用寄存器	AX、BX、CX、DX、SP、BP、DI、SI	EAX、EBX、ECX、EDX、ESI、EDI、EBP、ESP
段寄存器	CS、DS、SS、ES	CS、DS、SS、ES、FS、GS
段描述符寄存器	无	对程序员不可见
状态和控制寄存器	FLAGS 、IP	EFLAGS、EIP、CR0、CR1、CR2、CR3
系统地址寄存器	无	GDTR、IDTR、TR、LDTR
调试寄存器	无	DR0--DR7
测试寄存器	无	TR0--TR7

保护模式(Protected Mode) 是一种和 80286 系列及之后的 x86 兼容 CPU 操作模

式。保护模式有一些新的特色，设计用来增强多功能和系统稳定度，比如内存保护、分页、系统以及硬件支持的虚拟内存。大部分的现今 x86 操作系统都在保护模式下运行，包含 Linux、FreeBSD、以及 微软 Windows 2.0 和之后版本。需要指出的是，保护模式在增加这些新特性的同时，也带来了系统软件设计的复杂性

在 8086 时代，CPU 中设置了四个段寄存器：CS、DS、SS 和 ES，分别用于可执行代码段、数据段以及堆栈段。每个段寄存器都是 16 位的，对应于地址总线中的高 16 位。每条“访存”指令中的内部地址也都是 16 位的，但是在送到地址总线之前，CPU 内部会自动地把它与某个段寄存器中的内容相加。因为段寄存器中的内容对应于 20 位地址总线中的高 16 位(也就是把段寄存器左移 4 位)，所以相加时实际上是地址总线中的高 12 位与段寄存器中的 16 位相加，而低 4 位保留不变，这样就形成一个 20 位的实际地址，也就实现了从 16 位内存地址到 20 位实际地址的转换，或者叫“映射”。

段式内存管理带来了显而易见的优势：程序的地址不再需要采用内存物理地址进行编码了，调试错误也更容易定位了，更可贵的是支持更大的内存地址，程序员开始获得了自由。 然而，8086 系统很难从硬件层面对软件系统进行保护，在该环境下，应用程序可以直接对系统的任意内存地址（包括操作系统所在的区域）进行操作，所以该系统的安全漏洞几乎是显而易见的。

到了 80286 时代，它的地址总线位数增加到了 24 位，因此可以访问到 16MB 的内存空间。更重要的是从此开始引进了一个全新理念——**保护模式**。这种模式下内存段的访问受到了限制。访问内存时不能直接从段寄存器中获得段的起始地址了，而需要经过额外转换和检查。为了和 8086 兼容，80286 内存寻址可以有两种方式，一种是先进的保护模式，另一种是老式的 8086 方式，被称为实模式。Intel 选择了在段寄存器的基础上构筑保护模式，并且保留 16 位的段寄存器。不同的是，在保护模式下，段范围不再受限于 64K，可以达到 16MB（或者 80386 的 4GB）。这一下真正解放了软件工程师,他们不必再费尽心思去压缩程序规模，软件功能也因此迅速提升。

从 8086 的 16 位到 80386 的 32 位，看起来是处理器位数的变化，但实质上是处理器体系结构的变化，从寻址方式上说，就是从“实模式”到“保护模式”的变化。下面，我们详细讨论保护模式的引入带来的寻址方式、中断管理这两个方面的变化，以及相关的模式切换以及安全性问题。

2. 寻址方式的变化

1) 实模式下段的管理

实模式采用 16 位寻址模式，在该模式中，最大寻址空间为 1MB，最大分段为 64KB。由于处理器的设计需要考虑到向下兼容的问题，实模式也是我们今天接触到的大多数计算机在启动后处于的寻址模式。

8086 处理器地址总线扩展到 20 位，但算术逻辑运算单元（ALU）宽度即数据总线却只有 16 位，也就是说直接参与运算的数值都是 16 位的。为支持 1MB 寻址空间，8086 在实模式下引入了分段的方法。在处理器中设置了四个 16 位的段寄存器：CS、DS、SS、ES，对应于地址总线中的高 16 位。寻址时，采用以下公式计算实际访问的物理内存地址：

实际物理地址 = （段寄存器 << 4） + 偏移地址

这样，便实现了 16 位内存地址到 20 位物理地址的转换。

例如，这样的一句汇编代码：MOV AX，ES:[1200H]。在这里程序表示的意思是将 ES:[1200H]所指向的内存空间里的 16 位数据复制到 AX 寄存器中。ES 是指定的段寄存器，而 1200H 则是偏移量，假设寄存器 ES 中的值为 1000H，则根据以上的物理地址计算公式，

得到的实际访问地址为 11200H。

我们回顾一下实模式下程序的运行。程序运行的实质就是指令的执行，显然 CPU 是指令得以执行的硬件保障，而 CPU 是如何知道指令在什么地方呢？80x86 系列是使用 CS 寄存器配合 IP 寄存器的组合来通知 CPU 指令在内存中的位置。程序指令在执行过程中一般还需要有各种数据，80x86 系列有 DS、ES、FS、GS、SS 等用于指示不同用途的数据段在内存中的位置。程序可能需要调用系统的服务子程序，80x86 系列使用中断机制来实现系统服务。总的来说，这些就是实模式下一个程序运行所需的主要内容。

我们再来回顾一下实模式下的寻址方式。寻址方式一共有以下 8 种：

1. 立即数寻址　　例如：MOV AX, 1234H
2. 寄存器寻址　　例如：MOV AX, BX
3. 直接寻址　　例如：MOV AX, [1234H]
4. 寄存器间接寻址　　例如：MOV AX, [BX]
5. 基址寻址　　例如：MOV AX, [BX+100H]
6. 变址寻址　　例如：MOV AX, [SI+100H]
7. 基址加变址寻址　　例如：MOV AX, [BX+SI]
8. 带位移的基址加变址寻址　　例如：MOV AX, [BX+SI+100H]

纵然有这么多种的寻址方式，但实际上实模式的寻址本质上都是段基址左移 4 位加上偏移得到物理地址，如图 1-1 所示：

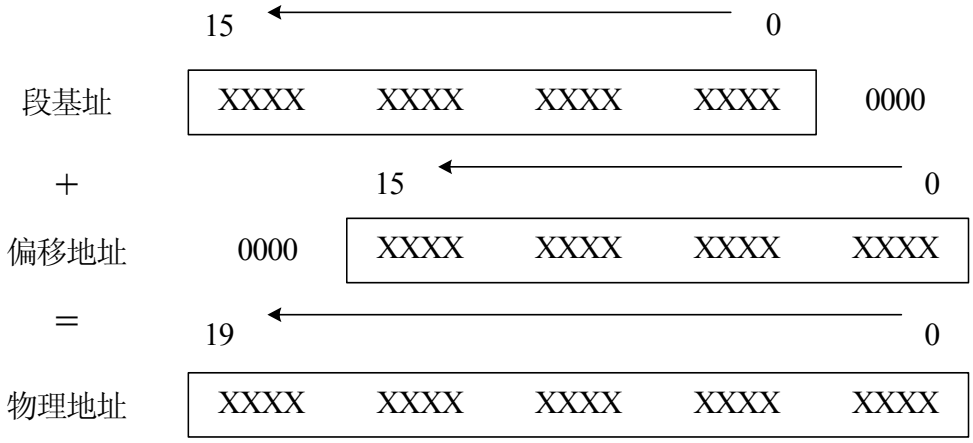

图 1-1 实模式的寻址

2) 保护模式下段的管理

保护模式的段式寻址相对于实模式而言较为复杂一些。在保护模式下，分段机制是利用一个称作段选择子的偏移量到全局描述符表中找到需要的段描述符，而这个段描述符中就存放着真正的段的物理首地址，然后再加上偏移地址量便得到了最后的物理地址。需要指出的是，在 32 位平台上，段物理首地址和偏移址都是 32 位的，实际物理地址的计算不再需要将段首地址左移 4 位了，直接相加即可，如果发生溢出的情况，则将溢出位舍弃。

在这里出现了两个新的词汇：段选择子、描述符。我们可以这样理解这个寻址过程，首先有一个结构体类型（称为段描述符，Descriptor），它有三个成员变量：段物理首地址、段界限、段属性，在内存中存在一个数组（称为全局描述符表，Global Descriptor Table）维护一组这样的结构体。段选择子（Selector）中存储的是对应的结构体在该数组中的下标，也就是索引，通过该索引从数组中找到对应的结构体，从而得到段的物理首地址，然后加上偏移量，得到真正的物理地址。

一般保护模式段式寻址可用 xxxx：yyyyyyyy 表示。其中 xxxx 表示索引，也就是段选

择子，是 16 位的；yyyyyyyy 是偏移量，是 32 位的。到哪里去寻找全局描述符表呢？80386 以及以后的处理器专门设计了一个寄存器 GDTR（Global Descriptor Table Register），专门用于存储全局描述符表在内存中存放的位置，当发生内存寻址与定位的时候，处理器通过该寄存器找到全局描述符表，并通过 xxxx 找到对应的描述符，进而得到该段的起始地址，并加上 yyyyyyyy 得到最终的物理地址。这个过程可以用图示 1-2 来描述。

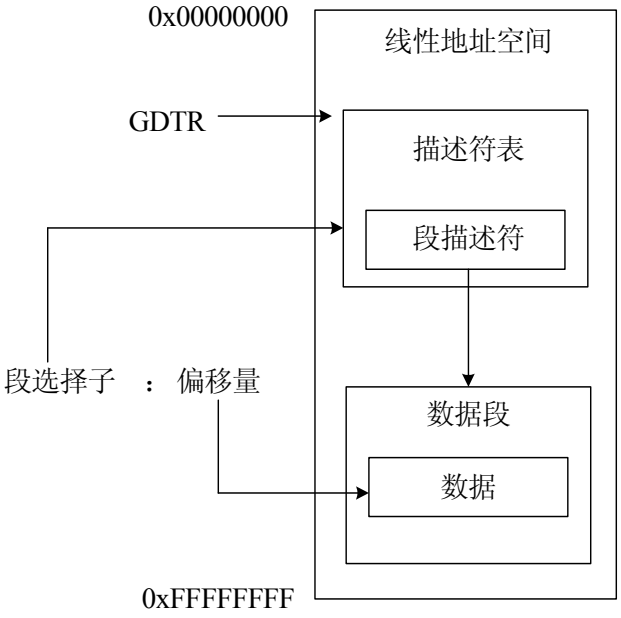

图 1-2 保护模式段式寻址

GDTR 寄存器有 48 位，其中有 32 位记录描述附表的物理地址，16 位记录全局描述符表的长度（该表占据的物理内存字节数），如图 1-3 所示。

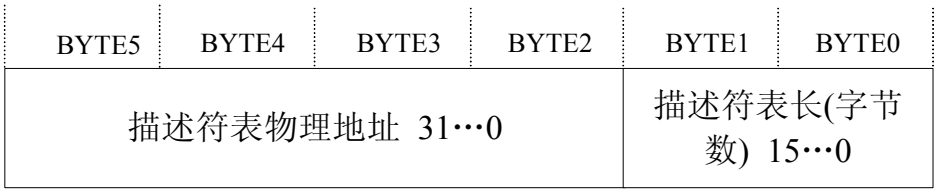

图 1-3 GDTR 位分布图

再来看看段描述符，段描述符实际上是一个占据 64 位内存（8 个字节）的结构体，如图 1-4 所示。

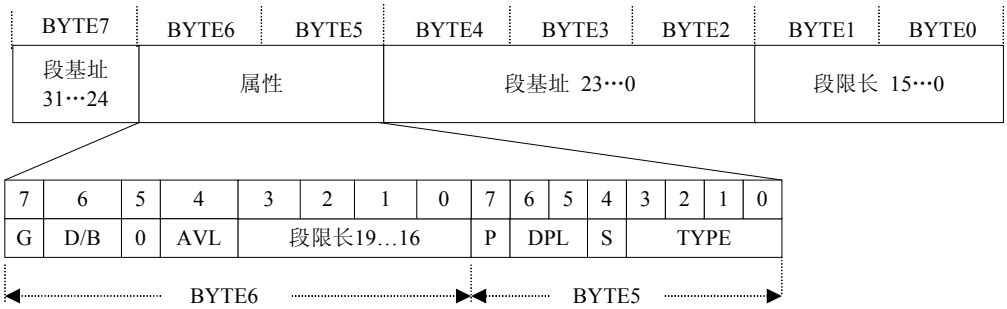

图 1-4 段描述符的结构

一个 64 位的段描述符包含了段的物理首地址、段的界限以及段的属性。在描述符中，段基址占 32 位，段限长占 20 位，属性占 12 位。

由图 1-4 可知，段基址为 2，3，4，7 字节，共 32 位。段限长为 0，1 以及 6 字节的低四位，共 20 位，段限长即段最大长度，与属性 G 共同确定。G = 0 时描述符中的 20 位段限长为实际段限长，最大限长为 1MB（0－FFFFFh）。G = 1 则 32 位段限长为描述符中的 20 位乘以 4KB，即段限长左移 12 位后加上 FFFH，最大限长为 4GB。可以看到，段首址的低 24 位被存放在第 2、3、4 号字节中，而高 8 位则存放在第 7 号字节中，将这四个字节合并便可以得到完整的段首地址。

下面我们将结合图 1-4 详细介绍一下段描述符中的段属性。

G：段限长粒度，G = 0 时，粒度为 1B，G = 1 时，粒度为 1 页（4KB）。

D/B：对于不同类型段含义不同。在可执行代码段中，这一位叫做 D 位，D = 1 使用 32 位地址和 32/8 位操作数，D = 0 使用 16 位地址和 16/8 位操作数。在向下扩展的数据段中，这一位叫做 B 位，B = 1 段的上界为 4GB，B = 0 段的上界为 64KB。在描述堆栈段的描述符中，这一位叫做 B 位，B = 1 使用 32 位操作数，堆栈指针用 ESP，B = 0 使用 16 位操作数，堆栈指针用 SP。

AVL：Available and Reserved Bit，通常设为 0。

P：存在位，P = 1 表示段在内存中。

DPL：描述符特权级，取值 0 ～ 3 共 4 级。0 特权级为最高，而 3 特权级为最低，表示访问该段时 CPU 所需处于的最低特权级，我们在后面会详细讨论特权级的问题。

S：描述符类型标志，S = 1 表示代码段或者数据段；S = 0 表示系统段（TSS、LDT）和门描述符。

TYPE：描述符类型，和 S 结合使用，可以表示的描述符类型有：代码段、数据段、TSS、LDT、中断门（Interrupt Gate）、陷阱门（Trap Gate）、调用门（Call Gate）、任务门（Task Gate）

其中，根据描述符类型标志 S 和 TYPE 可以确定描述符的类型。

当 S = 1 时，TYPE < 8 时，为数据段描述符。数据段都是可读的，不一定可写。如下表 1-2 所示：

1.2 描述符类型——数据段描述符

	TYPE				说明
十进制值		E	W	A	数据段
0	0	0	0	0	只读
1	0	0	0	1	只读，已访问
2	0	0	1	0	读/写
3	0	0	1	1	读/写，已访问
4	0	1	0	0	只读，向下扩展
5	0	1	0	1	只读，向下扩展，已访问
6	0	1	1	0	读/写，向下扩展
7	0	1	1	1	读/写，向下扩展，已访问

当 S = 1 时，TYPE ≥ 8 时，为代码段描述符。代码段都是可执行的，一定不可写。如下表 1.3 所示：

1.3 描述符类型——代码段描述符

	TYPE				说明
十进制值		C	R	A	代码段
8	1	0	0	0	只执行
9	1	0	0	1	只执行，已访问
10	1	0	1	0	执行/读
11	1	0	1	1	执行/读，已访问

12	1	1	0	0	只执行，一致
13	1	1	0	1	只执行，一致，已访问
14	1	1	1	0	执行/读，一致
15	1	1	1	1	执行/读，一致，已访问

S = 0 时，描述符可能为 TSS、LDT 和 4 种门描述符。如表 1.4 所示：

1.4 描述符类型——门描述符

十进制值	TYPE				说明
	11	10	9	8	
0	0	0	0	0	保留
1	0	0	0	1	16 位 TSS
2	0	0	1	0	LDT
3	0	0	1	1	16 位 TSS（忙）
4	0	1	0	0	16 位调用门（Call Gate）
5	0	1	0	1	任务门（Task Gate）
6	0	1	1	0	16 位中断门（Interrupt Gate）
7	0	1	1	1	16 位陷阱门（Trap Gate）
8	1	0	0	0	保留
9	1	0	0	1	32 位 TSS
10	1	0	1	0	保留
11	1	0	1	1	32 位 TSS（忙）
12	1	1	0	0	32 位调用门
13	1	1	0	1	保留
14	1	1	1	0	32 位中断门
15	1	1	1	1	32 位陷阱门

下面我将通过一个例子来加深对保护模式寻址方式的理解。

在这个程序开始执行的时候 cs 寄存器的值为 0.

```
    .set PROT_MODE_CSEG, 0x8
    .set CR0_PE_ON, 0x1
    lgdt    gdtdesc                          # 加载 GDTR 寄存器
    movl    %cr0, %eax
    orl     $CR0_PE_ON, %eax
    movl    %eax,  %cr0                      # 将 cr0 寄存器的末位置 1 标志进入保
护模式
    ljmp    $PROT_MODE_CSEG,  $protcseg      # 通过跳转使程序开始在保护模式下
执行
protcseg：
    movw    0x10,%ax
    movw    %ax,%ds                          # 给段选择子 ds 赋初始值
    movl    0xf000000,%ebx
    movl    0x20(%ebx), %eax
gdt:                                         # gdt 表的内容
```

```
    SEG_NULL                    # 空描述符
    SEG(STA_X|STA_R, 0x0, 0xffffffff) # 代码段
    SEG(STA_W, 0x0, 0xffffffff)       # 数据段
gdtdesc:
    .word   0x17                                # sizeof(gdt) – 1 （两个字节）
    .long   gdt                                 # address gdt （4 个字节）
```

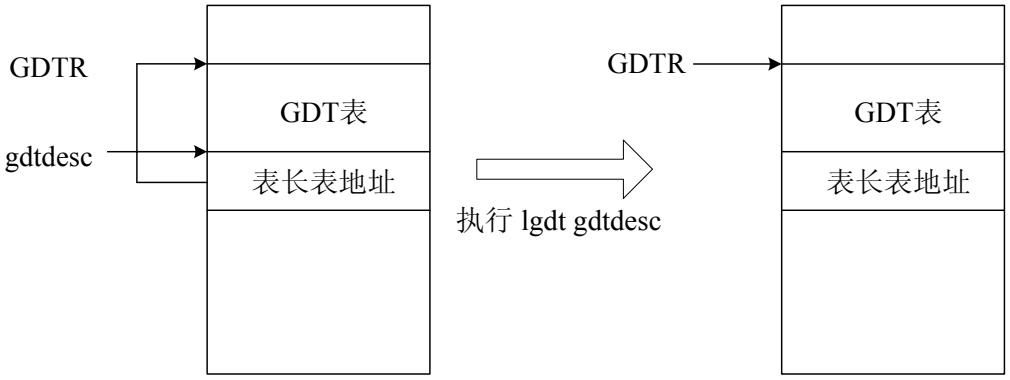

图 1-5 GDT 表的装入

以上这一小段程序展示了系统进入保护模式以及在保护模式中利用寄存器寻址的过程。首先 lgdt 指令将 GDT 表的地址和表长装入 GDTR 寄存器，可以看到在 gdtdesc 标识的地方存有一个字及一个双字，前者为 0x17 表示表的长度（字节数），后者是表的物理地址，在这里描述符表的首地址便是程序中 GDT 所标识的位置，这样 GDTR 中就存有之后寻址所需要的数据了，从图 1-5 可以看到这个过程。接着再将 CR0 的保护模式开启位打开，系统便进入了保护模式，开始采用保护模式的寻址模式进行地址的转换。这个时候，内存中有 GDT 的 3 个表项如图 1-6 所示。

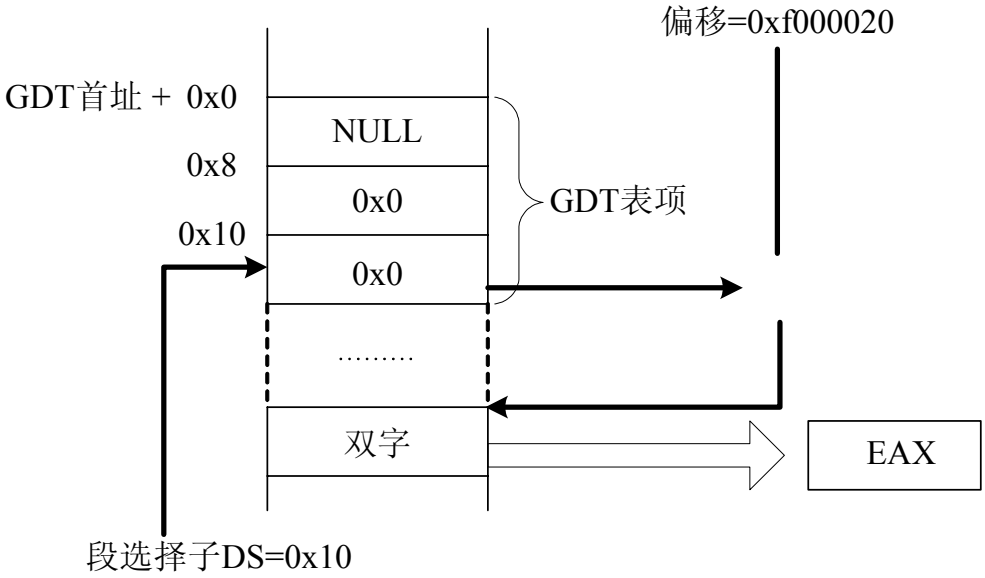

图 1-6 保护模式寻址过程示例

进入保护模式后系统立即执行了一个长跳转指令，由于是在保护模式中，所以 $PROT_MODE_CSEG 被当作段选择子，而$protcseg 仍是偏移地址。段选择子的值是 0x8，

于是对应的段描述符会是表中的第一项，即是 SEG(STA_X|STA_R, 0x0, 0xffffffff)这一项，0x0 表示段首地址是 0，所以最终得到的物理地址为 0+$protcseg，程序便会跳到 protcseg 所标识的位置来执行。之后会执行这样一句指令：movl　　0x20(%ebx), %eax，ebx 寄存器对应的段寄存器为 ds，而 ds 的值为 0x10，则如图 1-5 所示可知段基址为 0，于是物理地址是 0+0x20+0x f000000=0xf000020，内存中这个位置的一个双字会被复制到 eax 寄存器中。

另外在这里读者可能会有一个疑问，那就是既然在 32 位的保护模式下可以寻址 0 到 4G 的地址空间，要是如果机器上的内存不够 4G，若访问到高端地址岂不是会出错。其实这里能访问的到的 0 到 4G 的地址空间实际上是虚拟地址空间，在开启分页机制后，还要经过页表转换才能得到真实地址，而在开启分页之前系统一般会控制只访问低地址，这些问题到内存管理那一章(第四章)中我们会进行更深入的讨论。

3. 中断管理的变化

1) 实模式下中断的管理

无论是实模式还是保护模式，系统的中断机制都是通过一个叫做中断向量表的东西实现的。这个表就是一个存储一组向量的数组，其中每个向量就是一个数据结构，它包含 4 个字节，前两个字节表示 IP 的值，而后两个字节则是 CS 的值，于是这四个字节的组合共同指示了中断处理程序的第一条指令的位置 CS：IP。

在实模式下，中断向量表存放在物理内存开始的位置（0x0000---0x03ff），总共最多可以有 256 个中断向量，即对应最多 256 种中断。其中：

00h---04h 号中断向量为系统专用

08h---0fh 硬件中断(8259A 使用)

10h---1fh BIOS 使用

20h---3fh DOS 使用 如常用的 int 21h

40h---ffh　用户使用

2) 保护模式下中断的管理

在保护模式下中断向量表的基地址是存放在 IDTR 寄存器中的，因此中断向量表在内存中的位置实际上是由操作系统决定的，关于如何加载中断表，我们将会在第五章中进行详细的讲解。

在保护模式中，中断的类型可以分为两种。一种是硬件中断(Interrupts)，一种是软件中断(Exceptions)，又称为异常。

其中硬件中断在系统中是由外部事件所引起的，如：一次 I/O 操作的结束。其产生与 CPU 当前所执行的指令没有关系。从是否能够被屏蔽来划分，可将其分为两类：可屏蔽中断与不可屏蔽中断。其中前者由 CPU 的 INTR 引脚接收信号，后者由 NMI 引脚接收信号。我们可以通过 CLI 和 STI 指令来设置 EFLAGS 寄存器的 IF 位，如果这一位被清除，则 CPU 会禁止外部中断传递信号给 INTR 引脚，这样便屏蔽了可屏蔽中断，但是这对于 NMI 引脚不起作用，因此无法屏蔽不可屏蔽中断。

异常是在 CPU 执行指令期间遇到非法指令所产生的，根据是由是否可恢复和恢复点位置不同又可将异常划分为三种。它们是故障(Fault),陷阱(Trap)和中止(Abort)。其中 80386 认为故障是可以排除的，因此在 CPU 遇到引起故障的指令的时候，会保存当前的 CS 和 EIP 值，并转去执行故障处理程序。在故障排除后，执行 IRET 指令回到刚在引发故障的位置，重新执行刚才触发故障的指令。陷阱与故障的区别主要在于，在执行陷阱处理程序之前，系统所保存的 CS 和 EIP 的值代表引起陷阱的指令的下一条要执行指令所在的位置。而中止是

在系统发生严重错误时产生的。在引起中止后，当前执行的程序不能被恢复执行。

在这里我们只是对中断进行了一些简单的介绍，而在第五章中我们会对其做详细的讲解。

4. 重要寄存器以及模式的切换

保护模式与实模式的切换

在 x86 体系结构中，寄存器 CR0 是用来标志系统是处于实模式还是保护模式的。其中当 CR0 的第 0 位为 0 时，表示系统是处于实模式的，而为 1 时则是保护模式。下面的一段代码描述了实模式和保护模式之间的切换：

```
.set CR0_PE_ON,          0x1

movl      %cr0, %eax
orl       $CR0_PE_ON, %eax
movl      %eax, %cr0
```

可以看出，CR0 中的内容先被复制到 EAX 中，然后再与 0x1 相与于是便将末尾置 1，最后将修改后的值复制到 CR0 中便完成了模式的切换。

下面我们讲一个具体的程序示例，在这个程序的运行的过程中，系统会先处于实模式然后切换到保护模式。在这两个模式中，程序会分别向显存空间存放一个字符串，这样便可以在屏幕上显示出来。样读者便可以通过这个程序看到在实模式和保护模式下用不同的寻址方式将字符存放到的内存中。

```
  .set PROT_MODE_CSEG, 0x8              # 代码段选择子
  .set PROT_MODE_DSEG, 0x10             # 数据段选择子 r
  .set CR0_PE_ON,          0x1          # 保护模式启动标识

.globl start
start:
.code16                                 # 16 位模式
  cli                                   # 关中断
  cld                                   # 关字符串操作自动增加

# 设置重要数据段寄存器(DS, ES, SS).
  xorw      %ax,%ax                     # 将 ax 清零
  movw      %ax,%ds                     # 初始化数据段寄存器
  movw      %ax,%es                     # 初始化附加段寄存器
  movw      %ax,%ss                     # 初始化堆栈段寄存器

# 在实模式下通过向显存中写字节流在屏幕上打印"hello world"
  movw      $0xb800,%ax
  movw      %ax,%es
  movw      $msg1,%si
  movw      $0xbe2,%di
  movw      $24,%cx
```

```
    rep         movsb

    movw        $str,%si
    movw        $0xc04,%di
    movw        $28,%cx
    rep         movsb

# 打开 A20
seta20.1:
    inb         $0x64,%al                   # 等待到系统不忙
    testb       $0x2,%al
    jnz         seta20.1

    movb        $0xd1,%al                    # 输出 0xd1 到 0x64 端口
    outb        %al,$0x64

seta20.2:
    inb         $0x64,%al                   # 等待到系统不忙
    testb       $0x2,%al
    jnz         seta20.2

    movb        $0xdf,%al                    # 输出 0xdf 到 0x60 端口
    outb        %al,$0x60

# 从实模式切换到保护模式
    lgdt        gdtdesc
    movl        %cr0, %eax
    orl         $CR0_PE_ON, %eax
    movl        %eax, %cr0
# 通过长跳转使得程序在切换到保护模式的同时切换到 protcseg 处执行
    ljmp        $PROT_MODE_CSEG, $protcseg

.code32                                 # 32 模式
protcseg:
    # 在保护模式下设置数据段寄存器
    movw        $PROT_MODE_DSEG, %ax
    movw        %ax, %ds
    movw        %ax, %es
    movw        %ax, %fs
    movw        %ax, %gs
    movw        %ax, %ss

# 在保护模式下打印"hello world"
    movl        $msg2,%esi
```

```
    movl    $0xb8d22,%edi
    movl    $62,%ecx
    rep     movsb

spin:
    jmp spin

# GDT 表
.p2align 2                                  # GDT 表 4 字节对齐
gdt:
    SEG_NULL                                #  空表项
    SEG(STA_X|STA_R, 0x0, 0xffffffff)       #  代码段表项
    SEG(STA_W, 0x0, 0xffffffff)             #  数据段表项

gdtdesc:
    .word   0x17                            # gdt 表长度  - 1
    .long   gdt                             # gdt 表物理地址

#字符串
msg1:
    .byte 'i',0x7,'n',0x7,' ',0x7,'r',0x7,'e',0x7,'a',0x7,'l',0x7,' ',0x7,'m',0x7,'o',0x7,'d',0x7,'e',0x7
msg2:
    .byte 'i',0x7,'n',0x7,' ',0x7,'p',0x7,'r',0x7,'o',0x7,'t',0x7, 'e',0x7,'c',0x7,'t',0x7, 'e',0x7,'d',0x7,' ',0x7,
'm',0x7,'o',0x7,'d',0x7, 'e',0x7
str:
    .byte ':',0xc,' ',0xc,'h',0xc,'e',0xc,'l',0xc,'l',0xc,'o',0xc,' ',0xc,'w',0xc,'o',0xc,'r',0xc,'l',0xc,'d',0xc
```

例 1-1

可以看到，这段程序首先是运行在实模式下，在实模式下首先向物理地址为 0xb8be2 的内存区域存放一个字符串"in real mode"，再向物理地址为 0xb8c04 的内存区存放另一个字符串"：hello world"。由于段寄存器 es 在开始的时候被初始化为 0xb800，于是便可以 0xb800 为基址，以寄存器 di 中的 0xbe2 为偏移地址便可以在实模式下寻到 0xb8aa2 的内存区域，同理，也可以以 0xc04 为偏移地址寻址到 0xb8c04 的内存区域。在切换到保护模式后，寻址方式发生了变化，这个时候段寄存器里面存放的是段选择子，可以发现数据段寄存器 es 的值在切换到保护模式后便被初始化为 0x10，又可以看到 GDT 表的第二项的段描述符所给出的首地址为 0x0，于是将 es 作为段选择子，0xb8d22 作为偏移，则可以在保护模式下用“rep movsb”这样的语句将字符串"in real mode：hello world"放到物理地址为 0xb8d22 的显存区域。

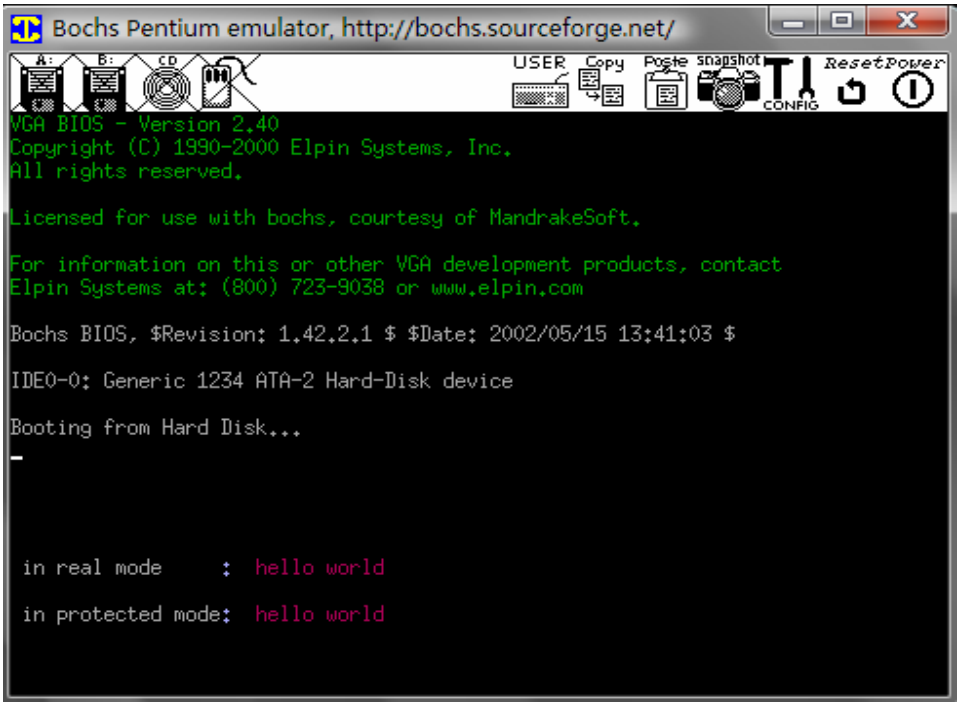

图 1-7 示例程序运行结果

上图便是这段代码的执行结果，可以看到，程序在实模式和保护模式下都以同样的方式在屏幕上输出了"hello world"。读者如果想要运行这段代码的话请参考 1.2 节。

5. 安全性问题

x86 平台 CPU 有 0、1、2、3 四个特权级，其中 level0 是最高的特权级，可以执行很多的特权操作，而 level3 则是最低的特权级，很多特殊的操作都不能在这个级别下进行。在实际中往往只需使用到 level0 和 level3 两个特权级，具体的就是操作系统内核运行时系统处于 level0，即 CS 寄存器的末两位为 00，而用户程序运行是系统是处于 level3 的，即 CS 的末两位为 11.

CPU 分级实际上有很多的好处。让操作系统运行在 level0 上，用户程序运行在 level3 上，这样通过页表权限的设置用户程序就不能随便访问内存中操作系统的代码，而操作系统则可以随意访问内存中的任何一个单元；另外，在 x86 架构下，有一些特权指令比如说执行 I/O 操作之类的需要 CPU 处于 level0 下才容许执行，这样便提高了安全性。这些问题在内存管理那一章会有更深入的讨论。

1.2Bochs 模拟器简介

我们即将介绍的这个实验是要完成一个简易的操作系统，而这个操作系统是跑在 x86 的硬件平台上的，于是在这里我们将用一个软件来模拟原始的 x86 的硬件平台，这个模拟器软件便是 Bochs。

Bochs 是一款模拟硬件平台的软件。软件可以再硬件平台上运行，同时也可以在软件模拟器中运行。因此在一台电脑中，你可以模拟任何其它的指令硬件系统出来，所以软件可以运行在虚拟硬件平台上。虽然这样软件运行的速度没有在真正的硬件平台运行快，但是这样的方式还是有很多优点的：

1) 你可以开发出任何其它类型的操作系统，不用考虑到现在你运行的系统。

2) 你可以运行不稳定的系统，而不用担心破环现在系统中的任何东西。
3) 对于操作系统开发者而言，可以不用重新启动你的电脑就可以调试。

1. Bochs 基本知识简介

1）什么是 Bochs

Bochs 是一款用来模拟 Intel x86 环境的模拟器。通过配置它可以实现 386、486、Pentium、Pentium II、Pentium III、Pentium 4 甚至是 x86-64 的模拟，还支持包括 MMX、SSEx 与 3DNow！这样的扩展指令。Bochs 能够解释从开机到重启机器的每一条指令，同时还可以对键盘、鼠标、显卡、硬盘、始终芯片、网卡等等外围设备进行模拟。因为 Bochs 模拟了整个 PC 的运行环境，于是在仿真环境中运行的软件会感到它好像就是运行在真实的机器上一样。这样 Bochs 便可以使很多的软件不加修改便运行在它所模拟的环境中。在 Bochs 中可以运行包括 Windows 95/98/NT/2000/XP/Vista、所有的 Linux 系列和所有的 BSD 系列在内的许多可以运行在 x86 架构下的操作系统。

Bochs 是用 C++程序语言编写的，而且它可以运行在许多不同的硬件平台上，包括像 x86、PowerPC、Alpha、Sun 以及 MIPS 之类的常用平台。无论在什么平台上运行，Bochs 都会模拟 x86 硬件平台。换一种说法便是，Bochs 的模拟完全不会受主机的机器指令的影响，这样的状况在有优势的同时也有它固有的劣势，这也是 Bochs 区别于其它很多像 plex86、VMware 之类的 x86 模拟软件的主要特点。因为 Bochs 会对每一条 x86 指令进行软件仿真，所以它可以在 Alpha 或者 Sun 的硬件架构上运行 Windows 的应用程序。然而 Bochs 的缺陷便是它的模拟效率比较低，为了准确的模拟 x86 的 CPU，Bochs 必须为了模拟一条 x86 指令而运行很多模拟指令，这样就使得虚拟的 x86 机器的运行速度会比真实的机器慢很多倍。商业的 PC 模拟器(VMware、Connectix 等等)通过一个叫做虚拟化的技术能够达到更快的模拟速度，但是它们既不是开源的也不能够运行在多平台上，于是我们在这里选择 Bochs 作为我们这个实验所使用的模拟器。

为了让程序能够在模拟的环境下运行，Bochs 需要去和主机上运行的操作系统进行交互。比如说，当你在 Bochs 显示窗口下敲下一个按键后，一个键盘事件会传到 Bochs 所模拟的键盘设备中；又如当需要从模拟的硬盘中读取数据的时候，Bochs 会从机器真实硬盘上的镜像文件中读取数据；当需要向网络中发送一个数据包的时候，Bochs 会使用真实机器的网卡。Bochs 这样的与底层操作系统进行交互的方式十分复杂，而且在不同的平台上 Bochs 模拟这些所需使用的指令往往是不一样。比如说，在 FreeBSD 操作系统中传送一个网络数据包和在 Windows 95 中是不一样的。正由于这个原因，Bochs 的某些特性仅仅只支持某些硬件平台，例如说，在 Linux 上，Bochs 可以模拟网卡来让在其之上运行的程序可以与外界网络交换数据，但是在 BeOS 上却不能，因为虚拟网卡设备与 BeOS 操作系统交互的代码还并没有完成。

2）Bochs 的用途

很难估计到底有多少人曾今使用过 Bochs，有个统计表明 SourceForge 网页上的最新版的 Bochs 有被下载过 150000 次。

Bochs 有很多用途，不同的人会利用它做不同的事。很多人利用 Bochs 在不需要两台机器的情况下在第二个操作系统里运行应用程序。在非 x86 的平台的上运行 Windows 应用程序是一个比较普遍的应用。另外，由于在 Bochs 中每一条硬件指令都可以被追踪到，它常常被用来调试新开发出来的操作系统。假如说你想自己写一个操作系统，你写了它的启动部分的代码但是却运行出错，这个时候你把它放到 Bochs 中去运行就可以通过 Bochs 的各种命令更清楚的看到在启动的过程具体都发生了什么。Bochs 调试器可以让你在运行操作系统的时候随时暂停运行来看看此时内存和 CPU 寄存器的状况。而且，当你想去了解哪一部分的程

序的运行最耗时间时，你可以用 Bochs 去测某一段代码被执行的频繁度。

Bochs 现在常常在操作系统教学中被用作一个教学的工具，学生使用 Bochs 并且修改它的代码去了解 PC 的硬件是怎么工作的。在工业中，Bochs 用来在新的硬件上支持旧的应用程序，并且在测试新的兼容 x86 的硬件的时候作为一个相关的参考模型。

3）Bochs 的特性

Bochs 拥有很多的特性，下面我们将用一个表格的形式来简单的说明 Bochs 的这些特性。

表 2-1 Bochs 特性

特性	是否支持	描述
配置脚本	是	Bochs 利用 GNU autoconf 来配置 Makefiles 与头文件，autoconf 帮助 Bochs 在多个平台上编译运行
386、486、Pentium 处理器的模拟	是	Bochs 只需要配置一下便可以仿真许多的 Intel 家族的硬件环境，目前有一些 Pentium 处理器的特性还不能实现例如像 MTRR、SMM 之类的
P6 系列和最新的 CPU 的模拟	是	Bochs 可以通过配置实现模拟 P6 系列的处理器，并且还支持可选择的 MMX 和 SSE 指令
Pentium4 模拟	未完成	少数 Pentium4 的特性还不能被仿真
x86-64 扩展的模拟	是	Bochs 能够通过配置来模拟 x86-64 平台并且实现 Intel 或者 AMD 的种种扩展
命令行的调试	是	强大的命令行调试器可以让你停止程序的运行并检查寄存器和内存，设置断点等等。不幸的是这个调试器目前还缺少对 x86-64 的支持
加强的 BIOS	是	实现 ELTorito、EDD v3.0 以及基本的 APM 特性。最新版本的 Bochs BIOS 支持 ACPI、SMM 和 SMP
VGA	是	在一个窗口实现彩色图形模拟
软盘	是	在所有的平台上支持软盘镜像，在 Unix 和 Windows 9x/NT/200/XP 下，Bochs 能够访问物理的软盘
多个 ATA 通道	是	可以模拟最多达 4 个 ATA 通道。最多实现 8 个 ATA/ATAPI 设备的模拟
硬盘	是	使用镜像文件来实现硬盘的模拟。读写物理硬盘在某些平台上是被支持的，然而为了安全考虑并不建议。硬盘最大可以达到 127GB。
CD-ROM	是	模拟 ATAPI-4/IDE 的 CD-ROM。这个模拟的 CD-ROM 能够在任何平台上读取 ISO 镜像文件。从 1.4 版本开始，Bochs 甚至可以从光盘启动或者从 ISO 镜像启动。
键盘	是	模拟北美标准键位分布的 PS/2 键盘
鼠标	是	模拟有 3 个按键和可选择的滚轮的 PS/2 接口或 USB 接口的鼠标
声卡	是	模拟声卡，可以将输出传到主机的声音设备
网卡	是	模拟兼容 NE2000 的网卡。在 Windows NT/2000、Linux、FreeBSD 和 NetBSD 中，Bochs 会从向操作系统中发送数据包同时也会从操作系统中接收数据，于是模拟系统便可以利用真实的网卡接入网络。
并行端口	是	在 Bochs1.3 版本中 Volker Ruooert 增加了并行端口的模拟。模拟系

		统中运行的程序向并行端口发送的数据会被存储到一个文件中国，然后 Bochs 会将其发送到真实的并行端口中去。
串行端口	是	可以在 GNU/Linux、NetBSD、OpenBSD、FreeBSD 和 MacOSX 中可以实现串行端口的模拟。最多可以模拟四个
游戏端口	是	模拟标准的 PC 游戏端口。目前仅仅在 Linux 和 win32 上支持与真实的游戏操纵杆相连
PCI	是	可以模拟大多数的 i440FX 的 PCI 芯片
USB	未完成	访问真实设备的能力还未能实现
插头	是	在 Linux、MacOS X、Solaris 和 Cygwin 上可被支持
16/32 位的寻址	是	16 或者 32 位的操作数、堆栈以及寻址
v8086/paging	是	虚拟 8086 模式包括 v8086 模式扩展与分页
PIC	是	主/从可编程中断控制器
动态指令翻译/虚拟化	不	因为 Bochs 是被设计成多平台的，所以它不能做到指令翻译和实现虚拟化的功能
仿真多核	是	Bochs 能够通过修改配置文件实现最多 8 核的仿真。这个功能目前还处于试验阶段。
复制和粘贴	是	取决于主机平台，屏幕上的文本可以被输出到剪贴板，同时也可以通过剪贴板复制到屏幕上。

2. 用 Bochs 来搭建模拟硬件环境

1）在 Bochs 中运行一个操作系统所需具备的基本要求

- 要有一个可执行的 Bochs 程序。
- 一个虚拟的 BIOS 镜像，我们将这个镜像叫作 BIOS-bochs-latest。
- 一个虚拟的 VGA BIOS 镜像，比如说像 VGABIOS-elpin-2.40 这样的文件。

初次搭建 Bochs 运行环境最简易的方法便是使用配置文件示例，便是 bochsrc-sample.txt 这样一个文件。Bochs 在启动的时候会根据一个路径去寻找它所需要的配置文件，所以我们首先将这个文件拷贝到相应的路径。接下来我们便要按照我们的需要去对这个配置文件进行相应的修改，你也许会想要去建立一个虚拟的硬盘，并且利用模拟的软盘或者光驱来在硬盘上安装一个操作系统，而要做到这些都必须在配置文件中设置相应的部分。

2）Bochs 配置文件

Bochs 使用一个叫做.bochsrc 的配置文件去去寻找硬盘镜像以及判断 Bochs 的模拟应该怎么进行。当你启动 Bochs 的时候，它会找到它相应的配置文件并分析它的脚本。下个面这一行语句是从某一个配置文件示例中截取出来的：

ata0-master：type=disk，path= "30M.sample"，cylinders=615，heads=6，spt=17

boot：disk

这段配置文件的脚本的意思是指定 Bochs 模拟一个接在 ata0 通道上硬盘，这个硬盘将由一个镜像文件来模拟，“path= "30M.sample"”告诉 Bochs 这个镜像文件的路径，“cylinders=615，heads=6，spt=17”告诉 Bochs 这个磁盘的柱面数、磁头数和扇区数。而之后的“boot：disk”则表明 Bochs 中运行的操作系统是从硬盘启动的。

Bochs 配置文件的格式非常的严格，所以在配置的时候要确定在正确的地方使用空格并且使用小写字母。可以看到，配置文件的很多行首先都会有一个关键字来表明这一行所配置的内容是什么，然后紧接着的便是一个冒号，再接着便是几组“variable=value”这样的语句，每一组语句之间用逗号隔开。在最简单的情况下，我们只需要赋值一个变量。

从 1.3 版本以后，我们可以再 Bochs 配置文件中使用环境变量，比如像：

floppya：1_44="$IMAGES/bootdisk.img"，status=inserted

boot：floppy

在 2.0 版本后，我们能够在配置文件中使用“#include”使得 Bochs 可以从其它文件读取的所需要的信息。现在我们能够将平台信息和安装缺省信息放到一个全局配置文件中，例如假如说我们将全局配置文件放在/etc 目录下，便可以在 Bochs 配置文件中以如下的方式引用它：

#include /etc/bochsrc

下面我们将逐个讲解一下常用的 Bochs 配置文件的配置选项。

- megs

例如：

megs：32

megs：128

这个选项是用来设置我们想要在 Bochs 中模拟的内存大小。缺省值为 32MB，然而很多操作系统需要比这个更多的内存。Bochs 最多支持模拟 2GB 的内存容量。

另外值得注意的是，由于受限于机器的内存容量，Bochs 在很多情况下甚至都无法提供 1GB 的内存。

- cpu

例如：

cpu：count=2，ips=10000000

该选项定义了模拟的 cpu 的各个参数，下面我们详细说明一下可能出现的各个选项：

count

count 表示处理器的个数，当 Bochs 编译的时候支持 SMP 的模拟的时候，还可以表示每个处理器中核的个数以及每个核上的线程数。当 Bochs 不支持 SMP 的时候 count 的值只能为 1.

quantum

在讲控制权交给另一个 cpu 之前处理器最多可以执行的指令数。该选项仅仅存在于 Bochs 支持 SMP 架构的时候。

reset_on_triple_fault

在发生三次错误后重启 CPU。

cpuid_limit_winnt

表示是否将 CPUID 的函数限制在 3 个以内。

msrs

定义用户 CPU MSRs 的路径

vendor_string

设定由函数 CPUID(0x0)返回的 CPUID 向量字符串。

ips

表示模拟的每秒钟可以执行的指令数。

ips 被用来标准化很多依赖于时间的模拟事件。例如改变 ips 的值便会影响 VGA 的更新速度。下表将会列出在一些不同机器上的标准的 ips 设置。

Bochs 版本	主频	主机/编译器	标准 ips
2.3.7	3.2Ghz	Intel Core 2 Q9770 with WinXP/g++ 3.4	50 to 55 MIPS
2.3.7	2.6Ghz	Intel Core 2 Duo with WinXP/g++ 3.4	38 to 43 MIPS
2.2.6	2.6Ghz	Intel Core 2 Duo with WinXP/g++ 3.4	21 to 25 MIPS

2.2.6	2.1Ghz	Athlon XP with Linux 2.6/g++ 3.4	12 to 15 MIPS
2.0.1	1.6Ghz	Intel P4 with Win2000/g++ 3.3	5 to 7 MIPS

- romimage

  例如：

  romimage: file=bios/BIOS-bochs-latest, address=0xe0000

  romimage: file=$BXSHARE/BIOS-bochs-legacy, address=0xf0000

  romimage: file=mybios.bin, address=0xfff80000

  romimage: file=mybios.bin

  ROM BIOS 决定了 PC 刚启动的时候在没有装入操作系统的时候需要执行的指令。通常情况下，我们可以使用一个叫做 BIOS-bochs-latest 先行编译好的二进制文件。ROM BIOS 在在相关设置缺省的情况下在开机的时候是被存放到内存的 0xe0000 处，并且大小为 128k。在之前的 Bochs 版本中 BIOS 通常被装载在内存的 0xf0000 处，并且大小为 64k。我们也可以使用环境变量$BXSHARE 去确定 BIOS 文件的位置。其中 BIOS 的起始地址是可选择的，可以通过计算 BIOS 镜像的大小得出。

- optromimage1，optromimage2，optromimage3，optromimage4

  例如：

  optromimage1：file=optionalrom.bin, address=0xd0000

  这个选项可以让 Bochs 可以加载最多 4 个可选的 ROM 镜像。

  注意要确保使用内存中的只读区域(C8000 到 EFFFF 之间)加载这些 ROM 镜像。这些可选 ROM 不能覆盖 ROM BIOS(F0000 到 FFFFF 之间)以及 VIDEO BIOS(在 C0000 到 C7FFF 之间)。

- vgaromimage

  例如：

  vgaromimage: file=bios/VGABIOS-elpin-2.40

  vgaromimage: file=$BXSHARE/VGABIOS-lgpl-latest

  vgaromimage: file=$BXSHARE/VGABIOS-lgpl-latest-cirrus

  这个选项规定了 VGA ROM BIOS 的镜像文件的位置(将被装载到内存的 0xC0000 处)。

- vga

  例如：

  vga: extension=cirrus

  vga: extension=vbe

  我们可以通过这个选项指定显示扩展的使用。当赋值’none’时，我们便使用标准的无扩展的 VGA。另外的一些可选的赋值像’vbe’表示要支持 VBE(需要 VGABIOS-lgpl-latest 作为 VGA BIOS 镜像)以及’cirrus’表示支持 Cirrus SVGA(需要 VGABIOS-lgpl-latest-cirrus 作为镜像)。

- floppya/floppyb

  例如：

  floppya: 2_88=a:, status=inserted

  floppya: 1_44=floppya.img, status=inserted

  floppyb: 1_2=/dev/fd0, status=inserted

  floppya: 720k=/usr/local/bochs/images/win95.img, status=inserted

  floppya: image=floppy.img, status=inserted

  floppya: type=1_44

  在这里，floppy 指的是软盘，其中 floppya 是第一个软盘，而 floppyb 是第二个。如果我

们想要系统从软盘启动，则软盘镜像的路径要指向一个可以从之启动的磁盘。在很多操作系统中，Bochs 可以直接从真实的软盘中读取数据。

又像诸如“2_88、1_44、1_2、720k”这样的数字是表示模拟软盘的类型。之后像诸如“=floppya.img”之类的便是表明镜像的位置，而“status”便是用来设定初始状态的，有弹出和插入两种状态，通常我们都会设定插入状态。参数“type”表明模拟的软盘不用指定镜像路径和初始状态。

- ata0，ata1，ata2，ata3

  例如：

  ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14

  ata1: enabled=1, ioaddr1=0x170, ioaddr2=0x370, irq=15

  ata2: enabled=1, ioaddr1=0x1e8, ioaddr2=0x3e0, irq=11

  ata3: enabled=1, ioaddr1=0x168, ioaddr2=0x360, irq=9

  这个选项可以用来开启 4 个 ata 通道。其中每个通道需有两个 io 基地址，并且 irq 必须要被指定。在缺省的情况下，ata0 与 ata1 会按照上面给出的赋值开启。

- ata0-master，ata0-slave，ata1-*，ata2-*，ata3-*

  例如：

  ata0-master: type=disk, path=10M.img, mode=flat, cylinders=306, heads=4, spt=17, translation=none

  ata1-master: type=disk, path=2GB.cow, mode=vmware3, cylinders=5242, heads=16, spt=50, translation=echs

  ata1-slave: type=disk, path=3GB.img, mode=sparse, cylinders=6541, heads=16, spt=63, translation=auto

  ata2-master: type=disk, path=7GB.img, mode=undoable, cylinders=14563, heads=16, spt=63, translation=lba

  ata2-slave: type=cdrom, path=iso.sample, status=inserted

  这个选项定义了所有连接在 ata 通道上的设备的类型和特性。

表 2-2 ata 设备配置选项

选项	内容	可能的赋值
type	连接的设备的类型	[disk | cdrom]
path	镜像文件的路径	
mode	镜像文件类型，仅仅只对模拟硬盘有效	[flat | concat | external | dll | sparse | wmware3 | wmware3 | wmware4 | undoable | growing | volatile]
cylinders	仅仅对模拟硬盘有效	
heads	仅仅对模拟硬盘有效	
spt	仅仅对模拟硬盘有效	
status	仅仅对模拟光盘有效	[inserted | ejected]
biosdetect	biosdetection 的类型	[none|auto]，[cmos](仅仅对连接在 ata0 上的硬盘有效)
translation	BIOS translation 的类型(仅对硬盘有效)	[none | lba | large | rechs | auto]
model	规定的设备 ata 命令所返回的字符串	

我们在配置的时候必须要设定连接的设备的类型。在 Bochs2.0 及以后的版本中，这个类型可以是 dick 或者 cdrom，即硬盘或者光盘。

我们还需要为模拟设备设置镜像文件、iso 文件或者物理光驱的路径。想要创建一个硬

盘镜像文件可以去使用 bximage。

在 Unix 中，我们可以使用真实的设备当做 Bochs 的硬盘，但是并不建议这样去做。在 Windows 中，要使用真实的设备却不是那么容易的事情。

- boot

  例如：

  boot: floppy

  boot: cdrom, disk

  boot: network, disk

  boot: cdrom, floppy, disk

  该选项规定了系统从哪启动以及启动的顺序。我们可以规定做多 3 个启动设备，这些设备可以是软盘、硬盘、光盘或者网络，即‘floppy’、‘disk’、‘cdrom’和‘network’。传统的‘a’和‘c’也是可以被支持的，比如像：boot　c

- floppy_bootsig_check

  例如：

  floppy_bootsig_check: disabled=1

  该选项屏蔽了从软盘启动的检查，在缺省情况下这个检查是需要进行的。

- log

  例如：

  log: bochsout.txt

  log: -

  log: /dev/tty　　(Unix only)

  log: /dev/null　　(Unix only)

  log: nul　　(win32 only)

  给出 Bochs 日志文件的路径。日志文件可以用来方便调试错误。如果我们不需要这个文件的话，可以将路径设置为”/dev/null”(Unix)或者”nul”(win32).

- debug/info/error/panic

  例如：

  debug: action=ignore

  info: action=report

  error: action=report

  panic: action=ask

  在模拟的过程中，Bochs 会遭遇到一些事件，而用户可能会对想要了解这些事件。这些事件按重要的程度由低到高被分为四种：debug、info、error 以及 panic。Debug 信息往往只在写 Bochs 代码或者试图确定一个错误的位置的时候才使用。每秒钟也许会有几千条的 debug 信息，所以开启 debug 信息时要慎重考虑。Info 信息会告知一些有趣的事件，这些事件并不会经常发生。当发现不应该发生的情况时 Bochs 会产生 error 信息，但这些 error 往往并不会对模拟产生破坏。Error 的一个例子便是，当在模拟环境中运行的软件产生一个非法的硬盘操作命令时 Bochs 便会报 error 信息。当出现 panic 信息的时候表明 Bochs 此时已经无法进行正常的模拟而且可能需要立即停止运行，出现 panic 的原因可能是配置的原因也可能是模拟的问题。

- debugger_log

  例如：

  debugger_log: debugger.out

  debugger_log: /dev/null　　(仅限 Unix)

debugger_log: -

给出一个日志文件的路径，Bochs 会在这个文件中输出 debugger 的信息。如果我们不需要这样的日志文件，则可以将路径设为’/dev/null’或者’-’。

- vga_update_interval

  例如：

  vga_update_interval: 50000 # default

  vga_update_interval: 250000

  表示显存在虚拟时间中多少微秒被更新一次。

- keyboard_serial_delay

  例如：

  keyboard_serial_delay: 250 # default

  一个字符数据从键盘传到控制器的时间(微秒)

- keyboard_paste_delay

  例如：

  keyboard_paste_delay: 100000 # default

  从尝试粘贴字符到字符传到键盘控制器的时间间隔(微秒)。这样的时间间隔是留给模拟环境中的操作系统来处理字符流的。缺省的设定是 0.1 秒，因为在 Windows 通常就是这样的。

- ips

  例如：

  ips: 2000000 # default

  ips: 10000000

  模拟的每秒钟执行的指令数。该选项通常不建议使用，因为 cpu 选项有相同的设置，因此建议在 cpu 选项中设置这个参数。

- clock

  例如：

  clock: sync=none, time0=local # Now (localtime)

  clock: sync=slowdown, time0=315529200 # Tue Jan 1 00:00:00 1980

  clock: sync=none, time0=631148400 # Mon Jan 1 00:00:00 1990

  clock: sync=realtime, time0=938581955 # Wed Sep 29 07:12:35 1999

  clock: sync=realtime, time0=946681200 # Sat Jan 1 00:00:00 2000

  clock: sync=none, time0=1 # Now (localtime)

  clock: sync=none, time0=utc # Now (utc/gmt)

  该选项定义了 Bochs 中时钟的参数，设定参数的格式如下：

  clock: sync=[none|slowdown|realtime|both], time0=[timeValue|local|utc]

  其中 sync 表示同步的信息。

  time0 表示模拟机器的启动时间。可以使用 time(2)系统调用所返回的时间。如果 time0 的值没有被设定或者被指定为 1 或者为’local’时，启动时间会被指定为当前机器上的真实时间。如果 time0 等于 2 或者等于’utc’，启动时间会被指定为当前的 utc 时间。

- mouse

  例如：

  mouse: enabled=1

  mouse: enabled=1, type=imps2

  mouse: enabled=1, type=serial

  mouse: enabled=0

假如 enabled 选项被设置为 0 则 Bochs 的图形化界面就不会制造模拟的鼠标事件。除非我们有特殊的原因需要激活鼠标的功能，否则建议在模拟的时候关闭这个功能。

通过设置 type 这个选项，我们可以选择模拟什么类型的鼠标，在这里默认值是’ps2’。其它可供选择的值有’imps2’、’serial’、’serial_wheel’以及’serial_msys’。

- private_colormap

例如：

private_colormap: enabled=1

该选项 enabled=1 时表示要求图形化界面使用非共享的图像设置。这样的图像设置会被使用在 Bochs 的窗口中。如果 enable=0 则 GUI 会使用一个共享的图像设置方案。

- keyboard_mapping

例如：

keyboard_mapping: enabled=0, map=

keyboard_mapping: enabled=1, map=gui/keymaps/x11-pc-de.map

enabled=1 表示需要对真实物理键盘与虚拟键盘之间进行重新匹配，enabled=0 表示不需要进行重新匹配。如果要重新匹配则需要给出键位匹配文件的路径。

下面我们介绍一个 Bochs 配置文件的示例：

#ROM 镜像路径

romimage: file=$BXSHARE/BIOS-bochs-latest, address=0xf0000

#CPU 个数为 1，且每秒钟执行 10000000 条指令

cpu: count=1, ips=10000000

#内存大小为 32MB

megs: 32

VGA ROM 镜像的路径

vgaromimage: file=$BXSHARE/VGABIOS-lgpl-latest

#无扩展的 VGA

vga: extension=none

#开启一个 ata 通道

ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14

#设定在 ata0 的主通道上接入硬盘设备并设定硬盘镜像的路径以及硬盘的参数

ata0-master: type=disk, mode=flat, path="./obj/kern/bochs.img", cylinders=100, heads=10, spt=10

#设定系统从硬盘启动

boot: disk

#设定系统时钟，启动时间规定为物理机器的当前真实时间

clock: sync=realtime, time0=local

#设定需要进行软盘启动检查

floppy_bootsig_check: disabled=0

#设定日志文件的路径

log: bochs.log

#规定事件处理方式

panic: action=ask

error: action=report

info: action=ignore

debug: action=ignore

#设定不需要 debug 信息文件
debugger_log: -
#设定并行端口
parport1: enabled=1, file="/dev/stdout"
#设定 VGA 的更新频率
vga_update_interval: 300000
#设定键盘延时时间
keyboard_serial_delay: 250
keyboard_paste_delay: 100000
#设定在 Bochs 图形界面中鼠标不可用
mouse: enabled=0
#图形化界面使用非共享的图像设置
private_colormap: enabled=0
设定虚拟键盘不需要与真实键盘进行重新匹配
keyboard_mapping: enabled=0, map=

例 3-1

Bochs 可以通过上述的这个配置文件来设置一个模拟的环境。

3. Bochs 调试命令

我们即将要进行的操作系统试验是运行在 Bochs 当中的，显然我们需要熟悉 Bochs 的一些调试命令这样才能够在试验的过程当中发现并修改代码的错误。在我们刚开始启动 Bochs 时，如果是使用“bochs”命令启动的，我们会看到如图 3-1 所示的的提示选项：

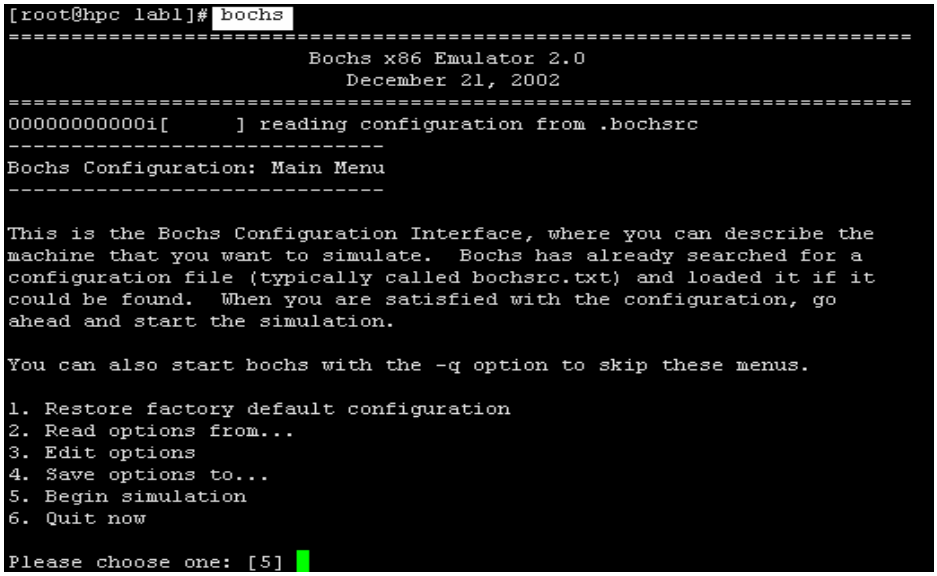

图 2-1 Bochs 启动选项

在选择选项 5 后我们便可以看到如图 3-2 所示的命令行 bochs：1>，或者我们使用“bochs –q”命令启动 Bochs 便可以直接开始模拟。

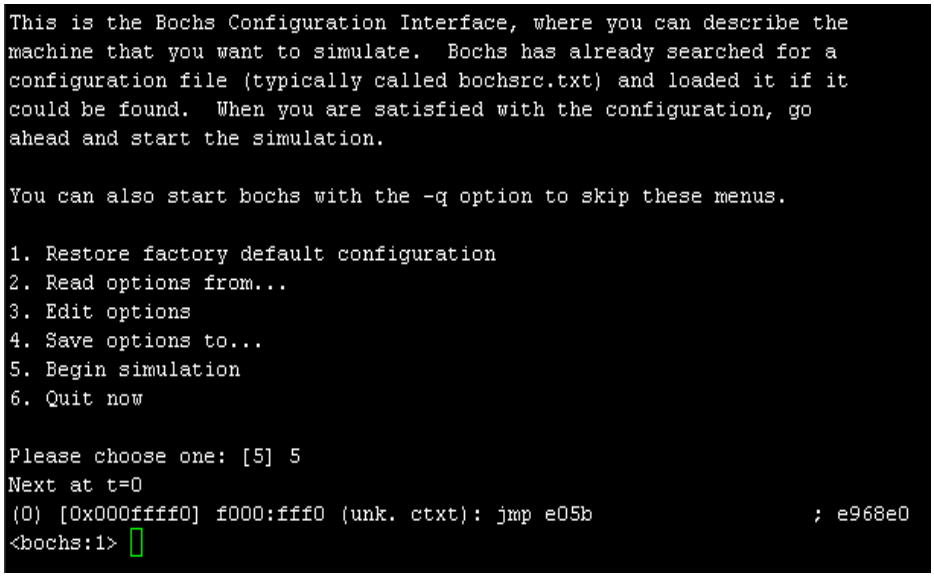

图 2-2 Bochs 命令行

从这里开始，我们便可以使用如下的 Bochs 调试命令

- 执行控制命令

c/continue 表示继续执行

s/step/stepi [count] 表示继续执行 count 条指令，如果没有设定 count 的值，则执行一条指令

Ctrl-C 停止执行，返回命令行

Ctrl-D 如果命令行此时是空的则退出 Bochs 模拟

q/quit/exit 退出调试和执行

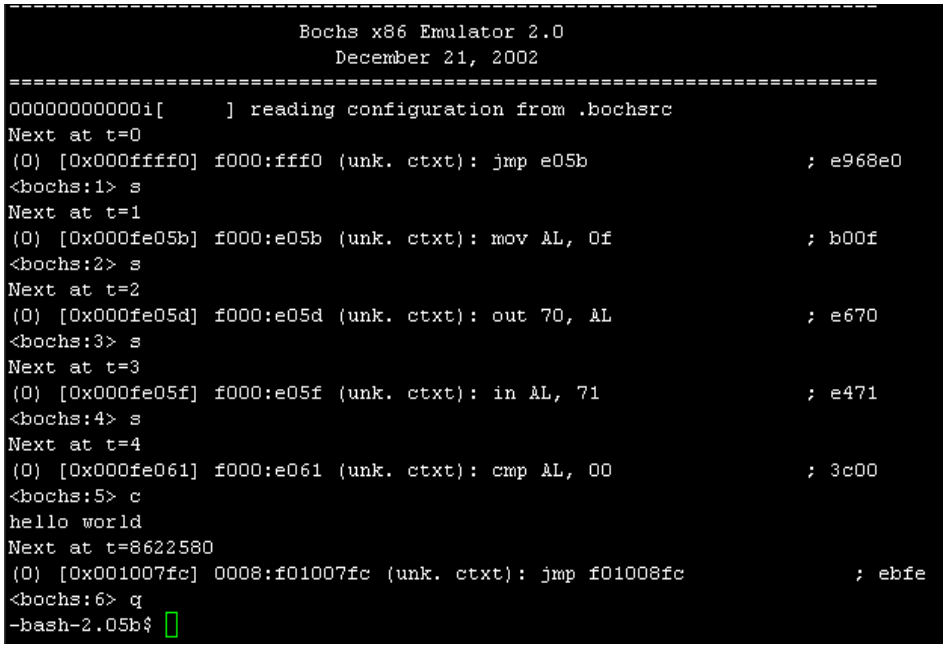

图 2-3 执行控制命令的使用

从图 3-3 中可以看到我们启动 Bochs 后便开始使用“s”命令单步执行，然后使用“c”命令继续执行打印出“hello world”，最后使用“q”退出模拟。

- 断点设置命令

vb/vbreak seg：off 用虚拟地址来设置断点，其中 seg 表示段基址，而 off 是偏移地址

lb/lbreak addr　　用线性地址来设置断点，addr 表示线性地址

b/break/pb/pbreak addr　　用物理地址来设置断点，addr 表示物理地址

info break　　显示当前所有的断点的状况

bpe n　　激活第 n 个断点

bpd n　　使第 n 个断点无效

d/del/delete　　删除第 n 个断点

假如在实模式下我们想要在物理内存 0x7c00 处设置一个断点，我们可以使用这的几种方式：vb 0x0:0x7c00、lb　0x7c00、b　0x7c00

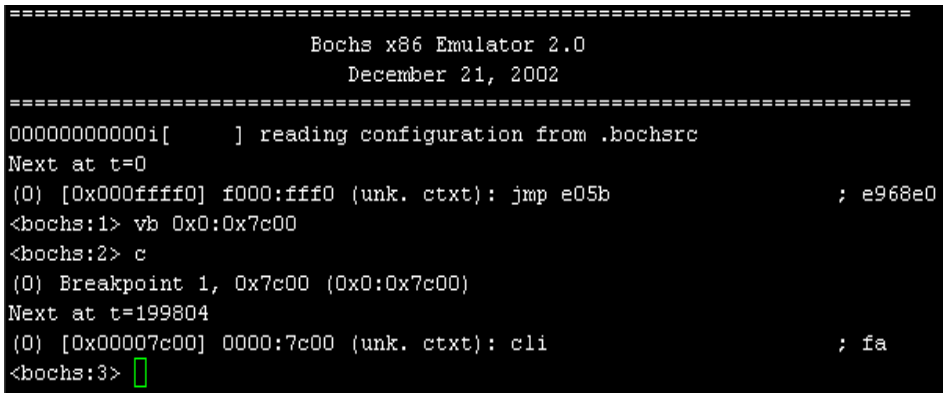

图 2-4 用虚拟地址设置断点

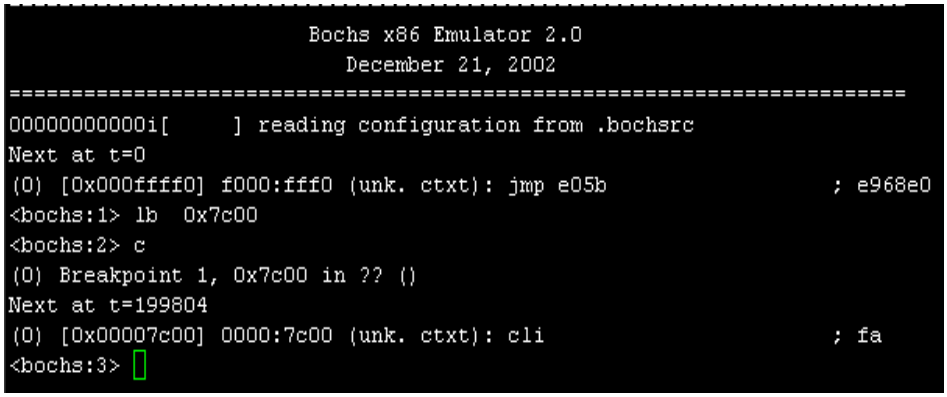

图 2-5 用线性地址设置断点

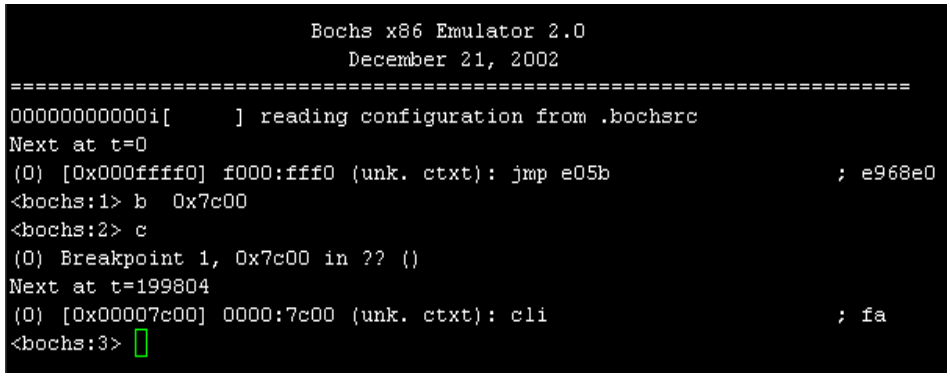

图 2-6 用物理地址设置断点

从以上的三副图中我们可以看到用三种不同的方式设置断点后程序都会在相同的时间点在相同的位置停止执行。

- 监视内存命令

watch r/read addr　　在物理地址 addr 处插入一个读内存监视符，即默认的情况下，如

果程序执行到某条指令需要在这个位置读取内存的时候便会自动停止

watch w/write addr　　在物理地址 addr 处插入一个写内存监视符，即默认的情况下，如果程序执行到某条指令需要在这个位置写内存的时候便会自动停止

watch　　显示所有当前的内存监视符

watch stop　　设定程序执行到监视符所对应的位置时，若满足读/写的条件则停止执行(默认设置)

watch continue　　设定程序执行到监视符所对应的位置时，若满足读/写的条件则还是继续执行

unwatch addr　　去除内存 addr 地址处的监视符

unwatch　　去除所有的内存读/写监视符

例如，若要在内存 0x7c00 处设置一个读内存的监视符，我们用如下的方式：

watch read 0x7c00

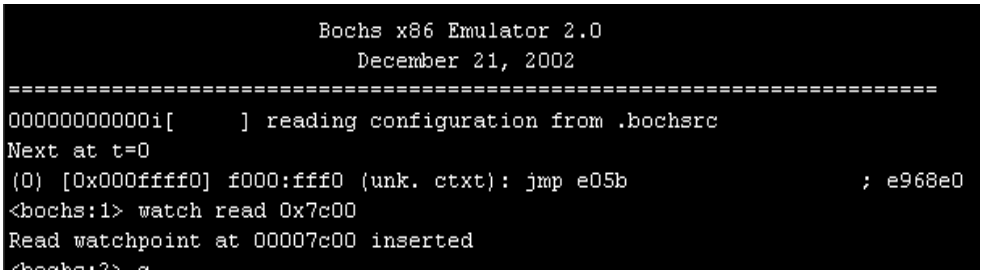

图 2-7 插入读内存的监视符

在上图中可以看到我们成功的插入了一个读内存的监视符，则当程序从内存的 0x7c00 的地址处读数据时便会自动的停下来。

又如当我们需要在内存 0x7c00 处设置一个写内存的监视符，我们用如下的方式：

watch write 0x7c00

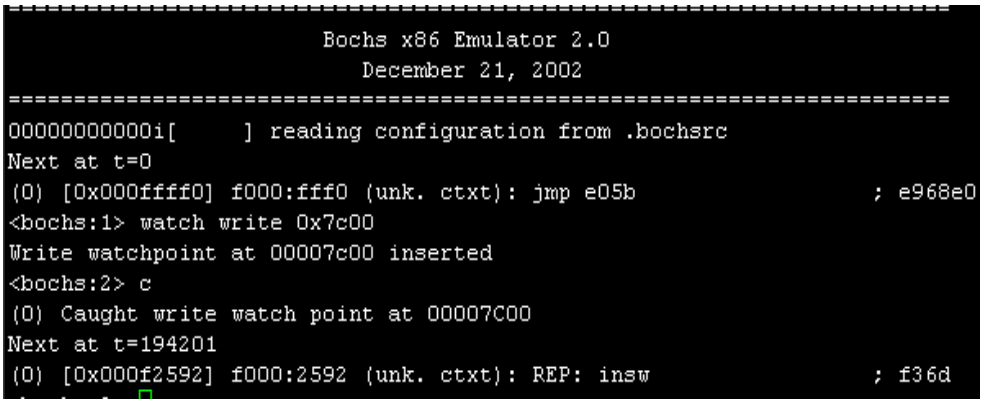

图 2-8 插入写内存的监视符

在图 3-8 中可以看到我们设置了一个写内存的监视符，于是当程序执行到 t=194201 时，由于此时需要向内存的 0x7c00 处写数据，所以程序这个时候停止执行。

- 查看内存命令

x [/nuf] addr　　查看当前内存某个地址的内容，其中 addr 代表线性地址

xp [/nuf] addr　　查看当前内存某个地址的内容，其中 addr 代表物理地址

我们可以看到在这里/nuf 是可选择的，其中 n、u、f 都有各自的含义：

n　　表示从所给的首地址开始显示内存多少个单元的内容

u　　表示显示内容每个单元的大小，每个单元的大小分为四种情况：

b 表示以字节为单位

	h 表示以字为单位(2 字节)
	w 表示以双字为单位(4 字节)
	g 表示以长字为单位(8 字节)
f	在屏幕上显示的形式
	x 以十六进制的形式显示
	d 以有符号十进制数的形式显示
	u 以无符号十进制数的形式显示
	o 以八进制的形式显示
	t 以二进制流的形式

在缺省的情况下，n 默认为 1，u 和 f 都是默认为上次使用的值，如果之前没有被用过则默认为 w 和 x。如果这三个可选参数都没有被使用，则我们不应该打'/'。同时 addr 也是可选择的参数，如果我们没有设定 addr 的值，则默认的值会是下一个地址。

例如，假若在内存的 0x7c00 处存放着一个双字 0xc031fcfa，则我们用如下的一些查询命令就会显示出不同的结果。

命令：x 0x7c00、x /b 0x7c00、x /2hx 0x7c00、x /2hd 0x7c00。

```
<bochs:2> x  0x7c00
[bochs]:
0x7c00 <bogus+0>:       0xc031fcfa
<bochs:3> x /b 0x7c00
[bochs]:
0x7c00 <bogus+0>:       0xfa
<bochs:4> x /2hx 0x7c00
[bochs]:
0x7c00 <bogus+0>:       0xfcfa  0xc031
<bochs:5> x /2hd 0x7c00
[bochs]:
0x7c00 <bogus+0>:       -774    -16335
<bochs:6> x /1bx 0x7c00
[bochs]:
0x7c00 <bogus+0>:       0xfa
<bochs:7> x /b
[bochs]:
0x7c01 <bogus+0>:       0xfc
```

图 2-9 查看内存内容

如图 3-9 所示以上的命令都会有相应的不同的结果显示出来。特别的，我们在使用了 x /1bx 0x7c00 的命令后紧接着使用 x /b 的话则会显示 0xfc。另外还有一个设置内存值的命令如下：

setpmem addr datasize val　将内存物理地址 addr 处 datasize 大小的单元设置为 val

例如：setpmem 0x7c00 1 0　即将 0x7c00 处的一个字节的单元的值赋为了 0，这个时候若在使用 x /1hx 0x7c00 命令则会显示 0xfc00。

- 查看信息的命令

r/reg/regs/registers	列出 CPU 的所有寄存器以及它们的当前值
fp/fpu	列出所有 FPU 寄存器和他们的当前值
mmx	列出所有 MMX 寄存器和他们的当前值
sse	列出所有 SSE 寄存器和他们的当前值
sreg	显示段寄存器以及它们的内容
creg	显示控制寄存器以及它们的内容
info cpu	列出所有 CPU 寄存器和他们的当前值

info eflags　　显示 eflags 寄存器的内容

info break　　显示当前的断点信息

- 操作 CPU 寄存器的命令

set reg = expr　　将 reg 寄存器设置为 expr 在目前我们只能够修改以下的寄存器的值：

eflags、eip、cs、ss、ds、es、fs、gs

例如：

set eax=2

- 指令追踪命令

trace on　　分解每一条可执行的指令，记录每一条导致异常的指令

trace off　　关闭指令追踪

- 其它的命令

ptime　　在屏幕上打印当前的时间(开始模拟后所经过的单位时间数)

sb delta　　插入一个时间断点，其中 delta 是相对于当前时间所过的单位时间数，例如像：sb 1000L，则当程序向后执行 1000 个单位时间后便会自动停下来，如图 3-10 所示：

```
Next at t=6300089
(0) [0x001007fc] 0008:f01007fc (unk. ctxt): jmp f01008fc              ; ebfe
<bochs:2> sb 1000L
Time breakpoint inserted. Delta = 1000
<bochs:3> c
(0) Caught time breakpoint
Next at t=6301089
(0) [0x001007fc] 0008:f01007fc (unk. ctxt): jmp f01008fc              ; ebfe
<bochs:4>
```

图 2-10 相对时间断点

sba time　　插入一个时间断点，其中 time 代表开始模拟后所经过的单位时间，例如像 sba 1000L，则当模拟开始后经过 1000 个单位时间后便会自动停下来，如图 3-11 所示：

```
Next at t=6300089
(0) [0x001007fc] 0008:f01007fc (unk. ctxt): jmp f01008fc              ; ebfe
<bochs:2> sb 1000L
Time breakpoint inserted. Delta = 1000
<bochs:3> c
(0) Caught time breakpoint
Next at t=6301089
(0) [0x001007fc] 0008:f01007fc (unk. ctxt): jmp f01008fc              ; ebfe
<bochs:4>
```

图 2-11 绝对时间断点

record filename　　将控制台所输入的命令记录到一个文件中，文件名为 filename

playback filename　　将文件中记录的命令读出来输入到控制台中

print-stack [num]　　打印堆栈中所存放的 num 个 16bit 的字在屏幕上。num 值在缺省的情况下为 16。在保护模式中仅仅是在堆栈基址为 0 的时候该命令才有效。

例如当我们输入：print-stack 3 时就会显示出从栈顶开始的 3 个字，如下图所示：


<bochs:1> print-stack 3
00000000 [00000000] 0000
00000002 [00000002] 0000
00000004 [00000004] 0000
<bochs:2>

图 2-12 查看堆栈内容