电子科技大学

计算机科学与工程学院

标 准 实 验 报 告

（实验）课程名称 计算机操作系统

电子科技大学教务处制表

学生姓名：董文龙　　学 号：2018081309003

指导教师：薛瑞尼

实验时间：2020 年 11 月 28 日

一、实验室名称：电子科技大学清水河校区主楼 A2-412

二、实验项目名称：虚拟内存综合实验

三、实验学时：4 学时

四、实验目的：

（1） 掌握计算机的寻址过程
（2） 掌握页式地址地址转换过程
（3） 掌握计算机各种寄存器的用法

五、实验原理：

5.1 物理地址，逻辑地址，虚拟地址，线性地址

物理地址：
物理地址就是内存单元的绝对地址，比如有一个 4G 的内存条插在电脑上，物理地址 0x0000 就表示内存条的第一个存储单元，0x0010 就表示内存条的第 17 个存储单元，不管 CPU 内部怎么处理地址，最终访问的都是物理地址。在 CPU 实模式下“段基址+段内偏移地址”就是物理地址，CPU 可以使用此地址直接访问内存。
逻辑地址：
操作系统或应用程序面对的存储单元地址的表示形式。分段存储管理方式把内存划分为多个逻辑段（代码段、数据段、堆栈段等），从而把不同的数据存储隔离开。这种情况下，用 “段起始地址+段内偏移地址” 这种形式来描述数据地址就是很自然的，这就是所谓的逻辑地址，它的描述形式是段号：偏移地址，例如实模式下 “MOV AX, [0x7C00]”，0x7C00 就是逻辑地址（或有效地址），但这条指令最终操作的物理地址是 DS*16+0x7C00。
虚拟地址：
Virtual Address，由于 Windows 程序时运行在 386 保护模式下，这样程序访问存储器所使用的逻辑地址称为虚拟地址。实际上因为我们现代程序中地址都是虚拟的，所以这里的虚拟地址和线性地址是等价了的。
线性地址：

线性地址（Linear Address）也叫虚拟地址(virtual address)，是逻辑地址到物理地址变换之间的中间层。在分段部件中逻辑地址是段中的偏移地址，然后加上基地址就是线性地址。

5.2 CPU 段式内存管理：逻辑地址转换为线性地址

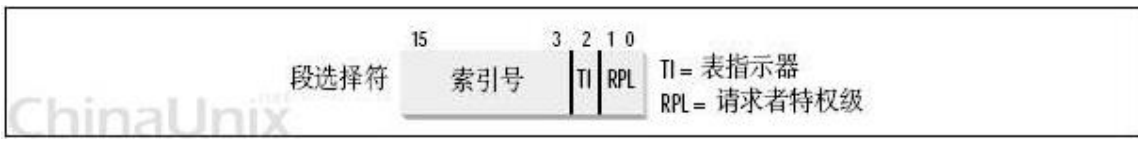

图 1 段选择符示意图

一个逻辑地址由两部份组成，段标识符: 段内偏移量。段标识符是由一个 16 位长的字段组成，称为段选择符，如图 1 所示，其中前 13 位是一个索引号。后面 3 位包含一些硬件细节，最后两位涉及权限检查。

索引号：可以看作是段的编号，也可以看做是相关段描述符在段描述符表中的索引位置。系统中的段表有两类：GDT 和 LDT。

段描述符：具体描述了一个段。在段表中，存放了很多段描述符。我们可以通过上面提到的段标识符的前 13 位，直接在段描述符表中找到一个具体的段描述符，也就是说，段标识符的前 13 位是相关段描述符在段表中的索引位置。

TI: T1=0，表示相应的段描述符在 GDT 中，T1=1 表示表示相应的段描述符在 LDT 中。

GDT：全局段描述符表，整个系统一个，GDT 表中存放了共享段的描述符， 以及 LDT 的描述符（每个 LDT 本身被看作一个段）

LDT：局部段描述符表，每个进程一个，进程内部的各个段的描述符，就放在 LDT 中。

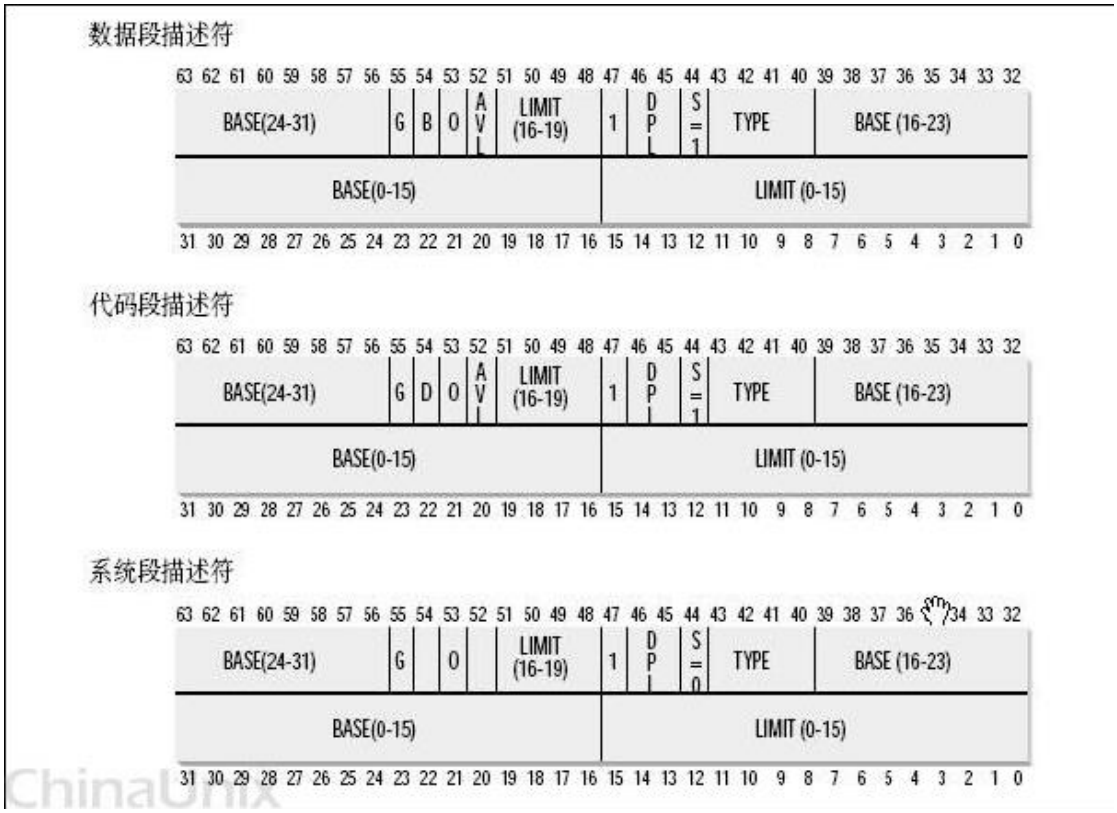

图 2 数据段描述符、代码段描述符、系统段描述符示意图

Base 字段：它描述了一个段的开始位置：段基址。

Base(24-31)：基地址的高 8 位

Base(16-23)：基地址的中间 8 位，

Base(0-15)：基地址的低 16 位。（这里的段基址，不是相应的段在内存中的起始地址，而是程序编译链接以后，这个段在程序逻辑(虚拟)地址空间里的起始位置。）

GDT 在内存中的地址和大小存放在 CPU 的 GDTR 控制寄存器中，而 LDT 则在 LDTR 寄存器中。具体如下图 3：

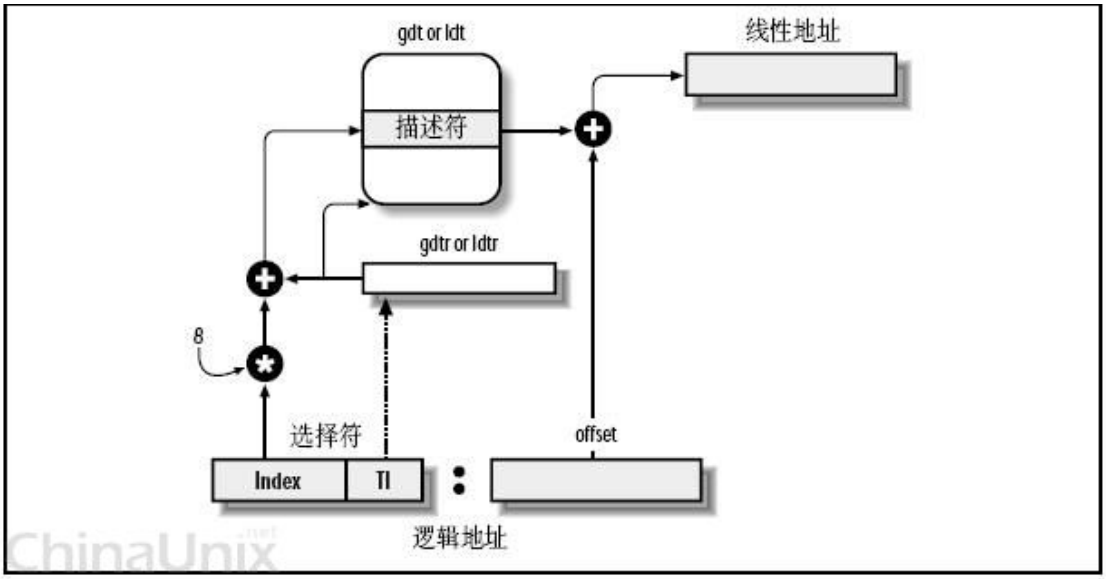

图 3 逻辑地址与线性地址的转换示意图

首先，给定一个完整的逻辑地址[段选择符：段内偏移地址]

1、看段选择符的 T1=0 还是 1，知道当前要转换是 GDT 中的段，还是 LDT 中的段，再根据相应寄存器，得到其地址和大小。我们就有了一个数组了。

2、拿出段选择符中前 13 位，可以在这个数组中，查找到对应的段描述符，这样，有了 Base，即基地址就知道了。

3、把 Base + offset，就是要转换的线性地址了。对于软件来讲，原则上就需要把硬件转换所需的信息准备好，就可以让硬件来完成这个转换了。

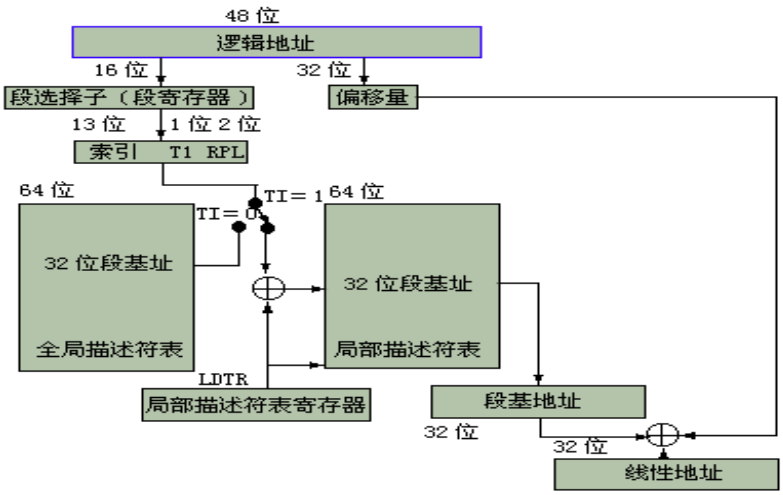

图 4 逻辑地址与线性地址的转换完整示意图

但是实际的情况并不是这么简单：linux 和 windows 的做法貌似是不同的。

5.3 CPU 页式内存管理：线性地址转化为物理地址

CPU 的页式内存管理单元，负责把一个线性地址，最终翻译为一个物理地址。从管理和效率的角度出发，线性地址被分为以固定长度为单位的组，称为页，例一个 32 位的机器，线性地址最大可为 4G，可以用 4KB 为一个页来划分，这页，整个线性地址就被划分为一个 tatol_page[2^20]的大数组，共有 2 的 20 个次方个页。这个大数组我们称之为页目录。目录中的每一个目录项，就是一个地址——对应的页的地址。

另一类“页”，我们称之为物理页，或者是页框、页桢的。是分页单元把所有的物理内存也划分为固定长度的管理单位，它的长度一般与内存页是一一对应的。这里注意到，这个 total_page 数组有 2^20 个成员，每个成员是一个地址（32 位机，一个地址也就是 4 字节），那么要单单要表示这么一个数组，就要占去 4MB 的内存空间。为了节省空间，引入了一个二级管理模式的机器来组织分页单元。如图 5：

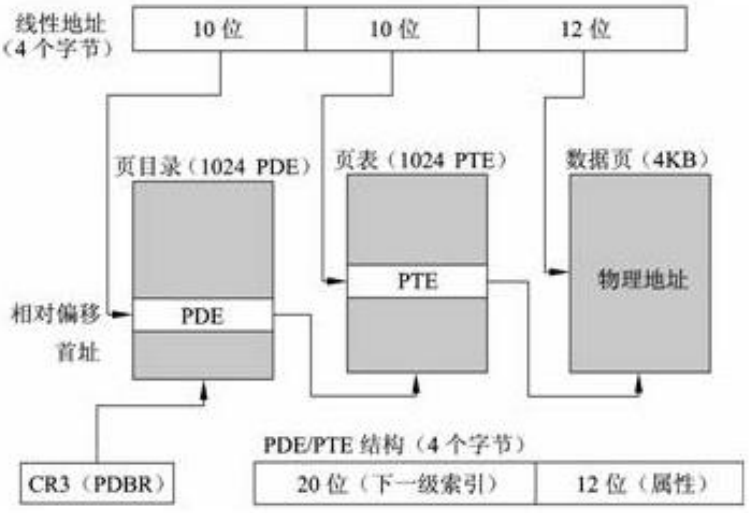

图 5 页式内存管理示意图

描述：

1、分页单元中，页目录是唯一的，它的地址放在 CPU 的 cr3 寄存器中，是进行地址转换的开始点。

2、每一个活动的进程，因为都有其独立的对应的虚似内存（页目录也是唯一的），那么它也对应了一个独立的页目录地址。——运行一个进程，需要将它的页目录地址放到 cr3 寄存器中，将别个的保存下来。

3、每一个 32 位的线性地址被划分为三部份，面目录索引(10 位)：页表索引(10 位)：偏移(12 位)

转换步骤：

1、从 cr3 中取出进程的页目录地址（操作系统负责在调度进程的时候，把这个地址装入对应寄存器）；

2、根据线性地址前十位，在数组中，找到对应的索引项，因为引入了二级管理模式，页目录中的项，不再是页的地址，而是一个页表的地址。（又引入了一个数组），页的地址被放到

页表中去了。
3、根据线性地址的中间十位，在页表（也是数组）中找到页的起始地址；
4、将页的起始地址与线性地址中最后 12 位相加，得到最终我们想要的物理地址；

六、实验内容：

通过手工查看系统内存，并修改特定物理内存的值，实现控制程序运行的目的。

七、实验器材：

PC 一台：
处理器：Intel(R)Core(TM) i7-8550U CPU @ 1.80GHz　2.00GHz
已安装的内存(RAM)：20.0GB
系统类型：64 位操作系统，基于 x64 的处理器
Linux 内核版本：0.11
Bochs 虚拟机版本：Bochs-2.6.11

八、实验步骤及结果：

1. 运行 Bochs-win64-2.6.11.exe，安装 Bochs 虚拟机，更改安装路径为 D:\Bochs\Bochs-2.6.11；
2. 安装完毕后，复制如图 6 所示实验所需文件至 Bochs 根目录下；

图 6 页式内存管理示意图

3. 修改配置文件中的路径，如图 7 所示

```
romimage: file="D:\Bochs\Bochs-2.6.11/BIOS-bochs-latest"
vgaromimage: file="D:\Bochs\Bochs-2.6.11/VGABIOS-lgpl-latest"
```

图 7 配置文件修改图片

4. 删除配置文件如下并保存，如图 8 所示

```
cpuid: family=6, model=0x03, stepping=3, mmx=1, apic=xapic, sse=sse2, sse4a=0, sep=1, aes=0, xsave=0, xsa
cpuid: vendor_string="GenuineIntel"
cpuid: brand_string="             Intel(R) Pentium(R) 4 CPU        "
```

```
pnic: enabled=0
```

```
keyboard_type: mf
keyboard_serial_delay: 250
keyboard_paste_delay: 100000
keyboard_mapping: enabled=0, map=
user_shortcut: keys=none
```

图 8 配置文件删除图片

5. 如图 9 所示，运行 bochsdbg.exe，点击 Load 加载配置文件，选择 mybochssrc-hd.bxrc 文件，再点击 Start 启动 Bochs 虚拟机；

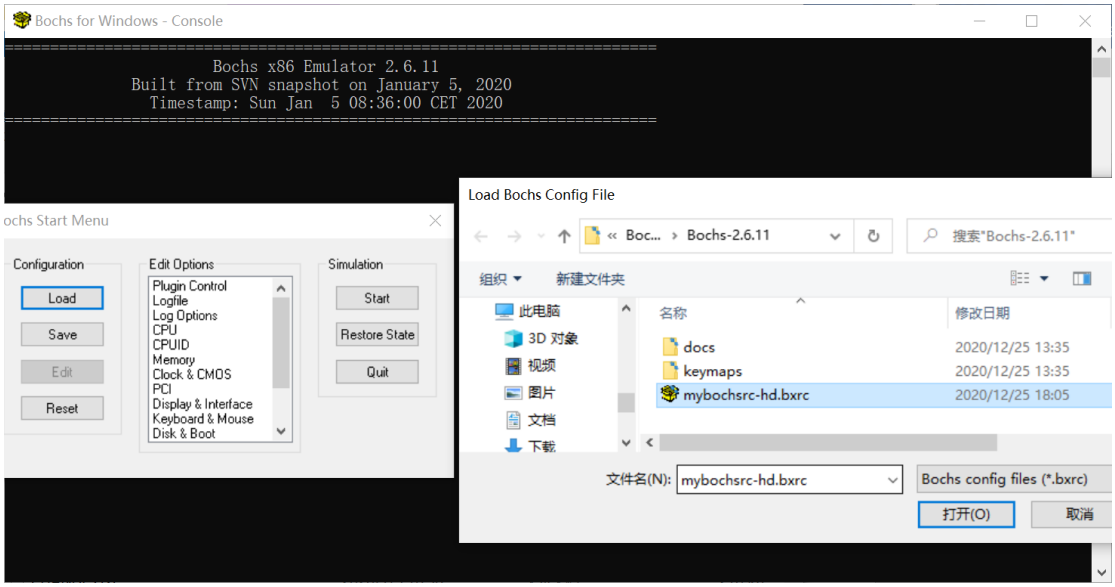

图 9 加载实验配置文件并启动 Bochs 虚拟机

6. bochs 虚拟机成功启动后，会出现两个窗口。一个窗口为 bochs 命令控制窗 Console，另一个为启动的 Linux 操作系统窗口 Display，图 10 所示。

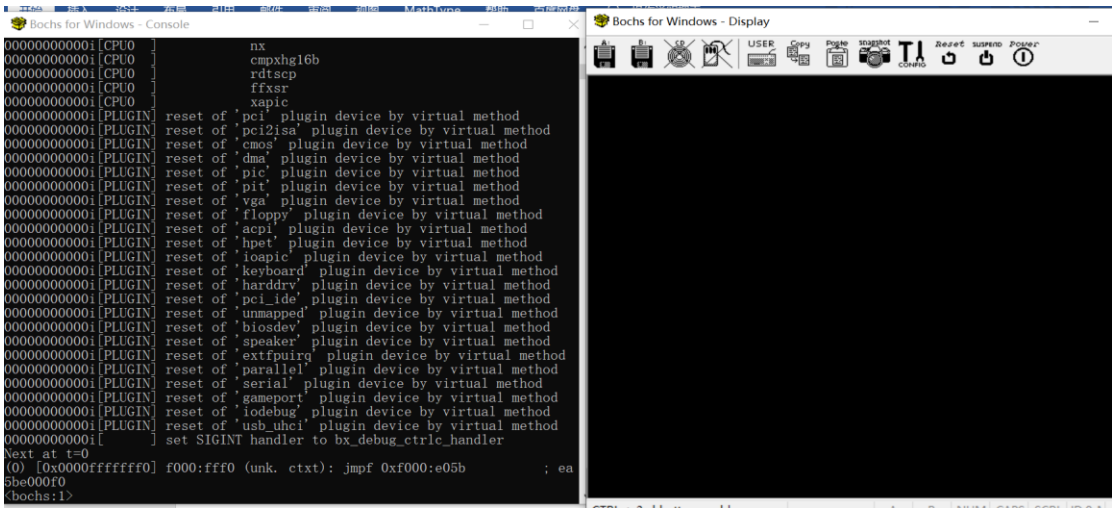

图 10 Console 窗口和 Display 窗口

7. 在 Console 窗口输入命令：c，加载 Linux 操作系统，如图 11 所示：

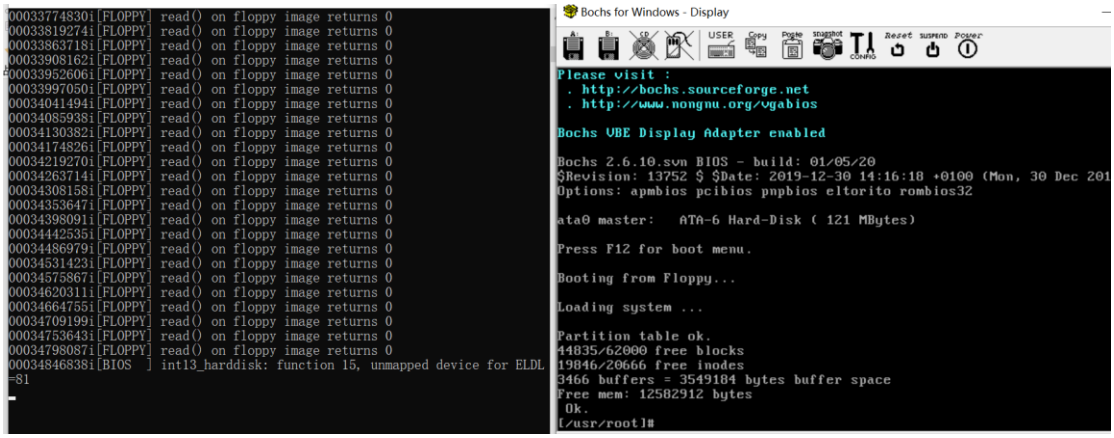

图 11 加载 Linux 操作系统

8. 在 Display 窗口输入命令：vi test.c，然后 Esc+a 命令输入实验程序，如图 12 所示，输入命令：Esc :wq，保存程序并退出；

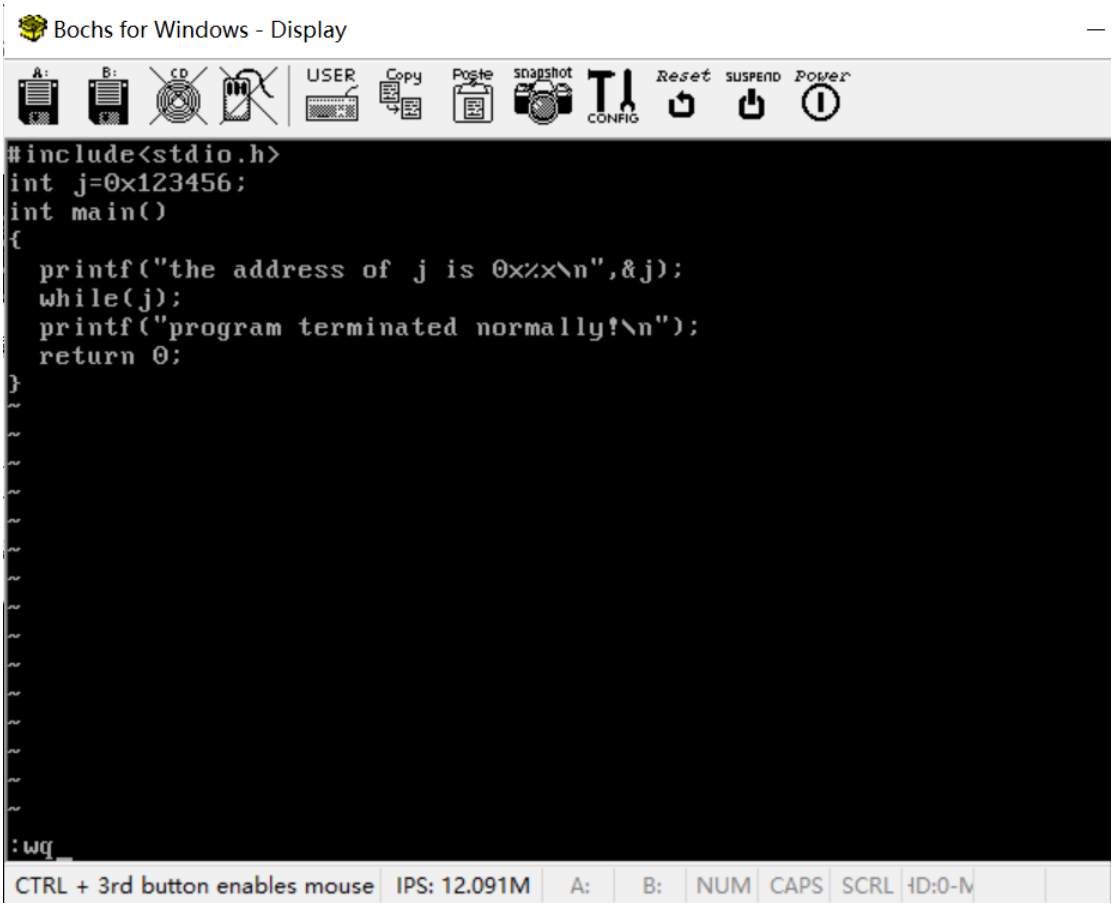

图 12 编写实验程序并退出

9. 输入命令：gcc -o test test.c，编译 test.c 程序，再输入命令：./test，运行程序，结果如图 13 所示，可以看出程序进入了死循环，并没有继续执行下一条语句了；

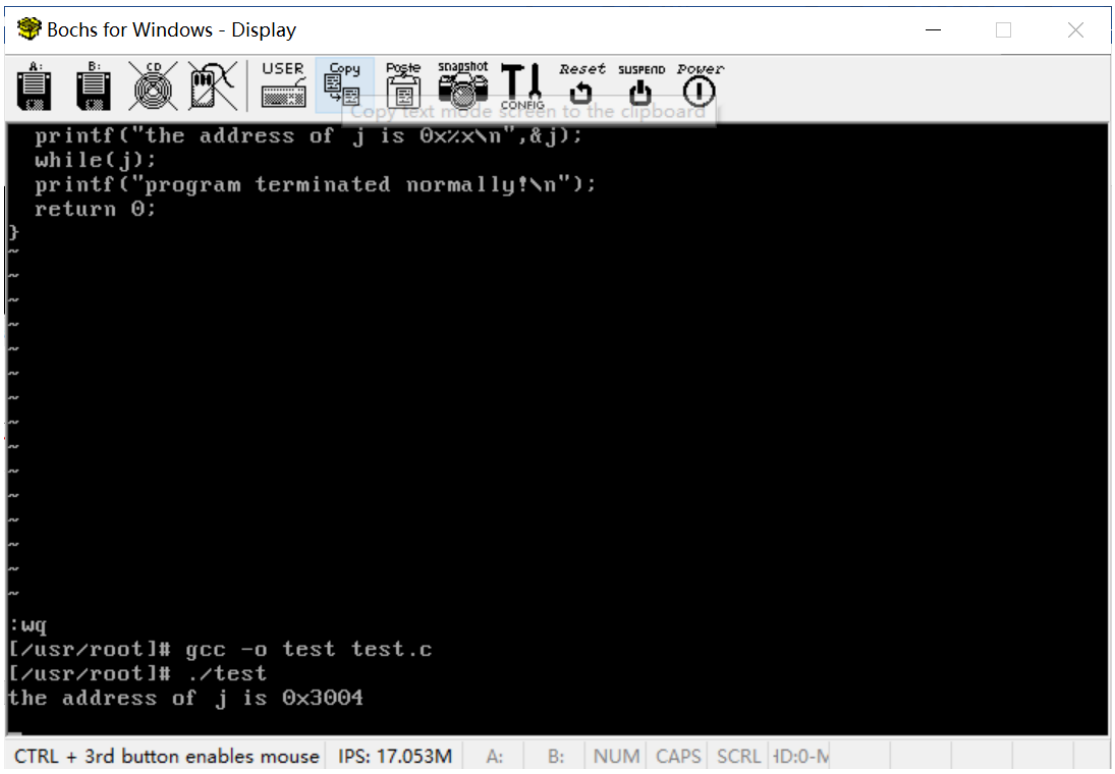

图 13 test.c 运行结果

10. 在 Console 窗口按下 Ctrl+C，中断当前程序运行，在 Console 窗口输入命令：sreg，查看段的具体信息，如图 14 所示；

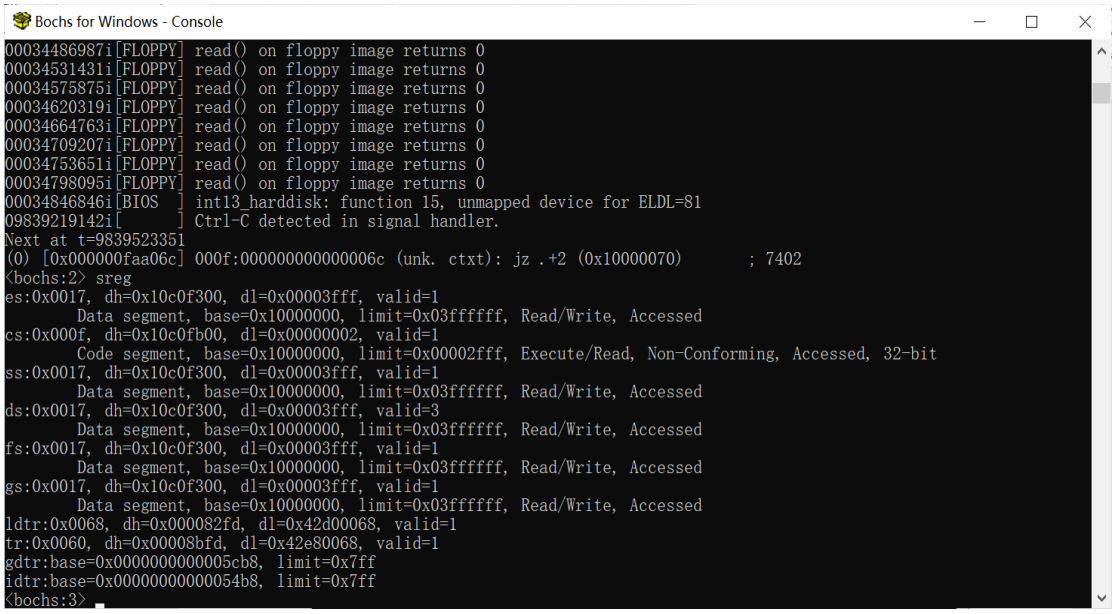

图 14 具体段的信息

11. 读 ds 段信息，根据 ds 段为 0x0017=0000 0000 0001 0111。其中，高 13 位代表索引号，索引号为 2。倒数第三位为 T1 位，T1=1，所以段描述符存放在局部段描述符表(LDT)中，应在局部段描述符表的第 3 项（起始号为 0）；
12. 读 LDTR 寄存器信息，其存放了 LDT 描述符在 GDT 中的位置。LDTR 为 0x0068=0000 0000 0110 1000。其中，高 13 位代表索引号，索引号为 13，所以 LDT 起始地址存放在 GDT 表的第 14 项（起始号为 0）；

13. 读 GDTR 寄存器信息，GDTR 为 0x5cb8，即 GDT 在内存中的起始地址为 0x5cb8，到每个段描述符由 8 个字节组成，输入命令：xp /2w 0x00005cb8+13*8，查看 GDT 中对应的表项，如图 15 所示。

```
<bochs:3> xp /2w 0x00005cb8+13*8
[bochs]:
0x0000000000005d20 <bogus+       0>:    0x42d00068      0x000082fd
```

图 15 GDT 中对应的表项

14. 其中 0x42d00068 为低位，0x000082fd 为高位，根据段描述符结构，第 16-31 位（即 42d0）对应基址的第 0-15 位，高 32 位的第 0-7 位（即 fd）对应基址的第 16-23 位，高 32 位的第 24-31 位(即 00)对应基址的第 24-31 位。故按此原理拼接地址，可以得到 LDT 的基址为： 0x00fd42d0。可以作如下验证：由之前的结果可得：因为段描述符在 LDT 表中的偏移为 2，输入命令：xp /2w 0x00fd42d0+2*8，得到的结果如图 16 所示。

```
Bochs for Windows - Console
09839219142i[      ] Ctrl-C detected in signal handler.
Next at t=9839523351
(0) [0x000000faa06c] 000f:000000000000006c (unk. ctxt): jz .+2 (0x10000070)       ; 7402
<bochs:2> sreg
es:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
        Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
cs:0x000f, dh=0x10c0fb00, dl=0x00000002, valid=1
        Code segment, base=0x10000000, limit=0x00002fff, Execute/Read, Non-Conforming, Accessed, 32-bit
ss:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
        Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
ds:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=3
        Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
fs:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
        Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
gs:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
        Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
ldtr:0x0068, dh=0x000082fd, dl=0x42d00068, valid=1
tr:0x0060, dh=0x00008bfd, dl=0x42e80068, valid=1
gdtr:base=0x0000000000005cb8, limit=0x7ff
idtr:base=0x0000000000054b8, limit=0x7ff
<bochs:3> xp /2w 0x00005cb8+13*8
[bochs]:
0x0000000000005d20 <bogus+       0>:    0x42d00068      0x000082fd
<bochs:4> xp /2w 0x00fd42d0+2*8
[bochs]:
0x0000000000fd42e0 <bogus+       0>:    0x00003fff      0x10c0f300
<bochs:5>
```

图 16 xp /2w 0x00fd42d0+2*8 的结果

查看 ds 段的段描述符信息，与 sreg 显示的 ds 段的 dl、dh 寄存器的值相同。

15. 由图 14 可知：ds 段的基址为 0x10000000，程序运行显示 j 的段内偏移地址为 0x3004，所以线性地址为：0x10000000+0x3004=0x10003004。将其用 0 补满 32 位，然后按照 10-10-12 比特的方式划分，位 0x40-0x03-0x04.即第一级页表内的索引为 0x40，第二级页表内的索引为 0x03，页内偏移为 0x04。
16. 输入命令：creg，如图 17 所示：

```
<bochs:5> creg
CR0=0x8000001b: PG cd nw ac wp ne ET TS em MP PE
CR2=page fault laddr=0x0000000010002fa8
CR3=0x000000000000
    PCD=page-level cache disable=0
    PWT=page-level write-through=0
CR4=0x00000000: cet pke smap smep osxsave pcid fsgsbase smx vmx osxmmexcpt umip osfxsr pce pge mce pae pse de tsd pvi vme
CR8: 0x0
EFER=0x00000000: ffxsr nxe lma lme sce
<bochs:6>
```

图 17 CR 寄存器的值

17. 寄存器 CR3 的值为 0，即页目录表的起始地址为 0。因此，对应页目录（PDE）地址为 0+64*4=0=0x100。输入命令：xp /w 0x100，查看页目录（PDE）的值，结果如图 18 所示；

```
<bochs:6> xp /w 0x100
[bochs]:
0x0000000000000100 <bogus+       0>:    0x00fa6027
<bochs:7>
```

图 18 PDE 的值

18. PTE 的值为 0x00fa6027。只取其前 20 位作为下一级的索引，即下一级的索引为 0x00fa6000。同理，对应页表（PTE）地址 0x00fa6000+3*4=0x00fa600c。输入命令：xp /w 0x00fa600c，查看页表（PTE）的值，结果如图 19 所示。

图 19 0x00fa600c 的值

19. PTE 的值为：0x00fa3067。同理只取其前 20 位作为下一级的索引，即下一级的索引为 0x00fa3000。因此，得到物理地址 0x00fa3000+4=0x00fa3004；
输入命令：xp /w 0x00fa3004，结果如图 20 所示。

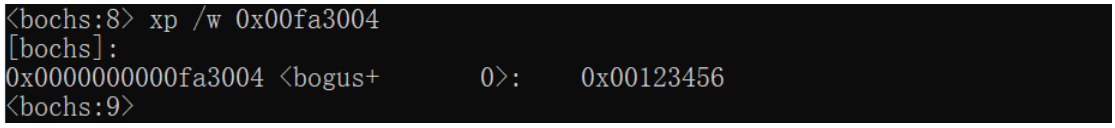

图 20 0x00fa3004 的值

20. 将结果进行对比，找到了 j 所在的正确物理地址 0x00fa3004
21. 输入命令：setpmem 0x00fa3004 4 0，将物理地址 0x00fa3004 的开始 4 个字节的值设置为 0，然后输入命令：c，继续运行。Display 窗口如图 21 所示。

图 21 0x00fa3004 内容改变后，display 窗口变化

程序成功执行了第二条输出语句并返回退出，该结果证明了以上实验步骤是正确的。
实验顺利完成。

九、总结及心得体会：

通过本次实验，我学习了操作系统的段页式内存管理机制，掌握了地址转换的过程，并且通过操作，能成功寻找到变量存储的具体位置，并对变量的数值成功进行修改。

十、对本实验过程及方法、手段的改进建议：

在 Load 配置文件的过程中，出现了好多情况的错误的 message，修改了很多次才弄成功，可以改正一下配置文件。

报告评分：

指导教师签字：