



中山大學  
SUN YAT-SEN UNIVERSITY

## 操作系统实验 3

### 从实模式到保护模式

实验课程：\_\_\_\_\_操作系统原理实验\_\_\_\_\_

实验名称：\_\_\_\_\_从实模式到保护模式\_\_\_\_\_

专业名称：\_\_\_\_\_计算机科学与技术\_\_\_\_\_

学生姓名：\_\_\_\_\_钟旺烜\_\_\_\_\_

学生学号：\_\_\_\_\_23336342\_\_\_\_\_

实验地点：\_\_\_\_\_实验楼 B203\_\_\_\_\_

实验成绩：\_\_\_\_\_

报告时间：\_\_\_\_\_2025 年 4 月 5 日\_\_\_\_\_

## Section 1 实验概述

### ●实验任务 1:

**1.1**、复现 Example 1，说说你是怎么做的并提供结果截图，也可以参考 Ucore、Xv6 等系统源码，实现自己的 LBA 方式的磁盘访问。

**1.2**、在 Example1 中，我们使用了 LBA28 的方式来读取硬盘。此时，我们只要给出逻辑扇区号即可，但需要手动去读取 I/O 端口。然而，BIOS 提供了实模式下读取硬盘的中断，其不需要关心具体的 I/O 端口，只需要给出逻辑扇区号对应的磁头（Heads）、扇区（Sectors）和柱面（Cylinder）即可，又被称为 CHS 模式。现在，同学们需要将 LBA28 读取硬盘的方式换成 CHS 读取，同时给出逻辑扇区号向 CHS 的转换公式。最后说说你是怎么做的并提供结果截图。

### ●实验任务 2:

复现 Example 2，使用 gdb 或其他 debug 工具在进入保护模式的 4 个重要步骤上设置断点，并结合代码、寄存器的内容等来分析这 4 个步骤，最后附上结果截图。

### ●实验任务 3:

改造“Lab2-Assignment 4”为 32 位代码，即在加载到保护模式后执行自定义的汇编程序。

## Section 2 实验步骤与实验结果

### ----- 实验任务 1 -----

#### ●任务要求:

#### Assignment 1

##### 1.1

复现 Example 1，说说你是怎么做的并提供结果截图，也可以参考 Ucore、Xv6 等系统源码，实现自己的 LBA 方式的磁盘访问。

在这个 lab 的 `src` 文件夹中，提供了 makefile 文件，同学们可以自行学习通过 `make` 的方法，更快捷地完成编译、运行任务。

##### 1.2

在 Example1 中，我们使用了 LBA28 的方式来读取硬盘。此时，我们只要给出逻辑扇区号即可，但需要手动去读取 I/O 端口。然而，BIOS 提供了实模式下读取硬盘的中断，其不需要关心具体的 I/O 端口，只需要给出逻辑扇区号对应的磁头（Heads）、扇区（Sectors）和柱面（Cylinder）即可，又被称为 CHS 模式。现在，同学们需要将 LBA28 读取硬盘的方式换成 CHS 读取，同时给出逻辑扇区号向 CHS 的转换公式。最后说说你是怎么做的并提供结果截图。

参考资料:

- CHS 和 LBA 的换算小记
- int 13h 中断

## ●思路分析:

实验 1 的任务为复现 Example1, Example1 的主要任务为加载 bootloader. 我们知道操作系统启动后最后只会自动加载 MBR(512 字节)到内存中运行, 然而, 512 字节能够完成的工作非常有限, 因此为了使操作系统能够访问更大的硬盘空间来执行更大的程序, 我们要在 MBR 上实现将一段程序从外存加载到内存中, 即实现读写硬盘。Example1 实现的便是通过 MBR 实现读写硬盘, 加载 bootloader 中的程序, 加载成功后在 qemu 屏幕上打印字符 “run bootloader”。

读写硬盘有两种方式, 一种为在实模式下利用 BIOS 中断来读取硬盘的方式, 称为 CHS 模式。另一种是通过逻辑块寻址模式, 简称为 LBA(Logical Block Addressing,)模式, LBA 模式主要利用计算机会在我们访问硬盘时会自动地将我们给出的逻辑块转换成对应的磁头、磁道和柱面, 最后使用计算出来的磁头、磁道和柱面来访问。因此实现 LBA 的关键在于设置 I/O 端口实现处理器和外围设备数据的交换。

实验 1.1, 即 Example1 所实现的代码聚焦于实现 LBA 模式, 而实验 1.2 则要求通过 CHS 模式实现硬盘的读写。

### 实验 1.1

使用 LBA 读取硬盘的方式如下:

- 1、设置起始的逻辑扇区号
- 2、将要读取的扇区数量写入 0x1F2 端口
- 3、向 0x1F7 端口写入 0x20, 请求硬盘读
- 4、等待其他读写操作完成
- 5、若在第 4 步中检测到其他操作已经完成, 那么我们就可以正式从硬盘中读取数据

根据以上 LBA 模式读取硬盘的 5 个步骤, 我们依次编写对应的模块代码, 最后整合便可以得到 MBR 中使用 LBA 模式读写硬盘的程序

### 实验 1.2

使用 CHS 模式读取硬盘, 要求在实模式下利用中断来实现。通过查询中断表, 可以利用 int 13h 中断来使用直接磁盘服务, 因为我们本次实验只需要把已经存入扇区的数据读出来, 故只用使用 AH = 02H 功能号。

功能 02H 下的入口参数有: AL=扇区数; CH=柱面; CL=扇区; DH=磁头; DL=驱动器, 00H~7FH: 软盘; 80H~0FFH: 硬盘; ES:BX=缓冲区的地址。所以要利用此功能实现读硬盘, 只需要给出逻辑扇区号对应的磁头 (Heads)、扇区 (Sectors) 和柱面 (Cylinder)。逻辑扇区号向 CHS 的转换公式如下所示:

## LBA和CHS换算关系

类比十进制计数方法，可以给出从CHS计算LBA的公式：

$$\text{LBA} = (\text{S} + \text{H} * \text{BPB\_SecPerTrk} + \text{C} * \text{BPB\_NumHeads} * \text{BPB\_SecPerTrk}) - 1$$

进而推导出从LBA计算CHS的公式：

$$\text{S} = (\text{LBA} + 1) \% \text{BPB\_SecPerTrk}$$

$$\text{H} = (\text{LBA} + 1) / \text{BPB\_SecPerTrk} \% \text{BPB\_NumHeads}$$

$$\text{C} = (\text{LBA} + 1) / \text{BPB\_SecPerTrk} / \text{BPB\_NumHeads}$$

【LBA+1可以表示第几个扇区，因为LBA是从0开始编号的】

有上述计算方法我们可以得出对应的磁头（Heads）、扇区（Sectors）和柱面（Cylinder），再设置对应寄存器的数值，即可实现 CHS 模式读取硬盘。

### ●实验步骤：

#### 实验 1.1

实验 1.1 的代码已经在参考资料里给出，下载到虚拟机中直接进行执行命令操作即可。

注意到这个 lab3 中的 src 文件里已经包含了一个 Makefile 文件，因此我们可以直接在 terminal 中通过 make 操作来实现.asm 文件转化为二进制文件并将其存入 hd.img 硬盘以及使用 qemu 启动操作系统的操作。

Makefile 文件内容如下：

```
build:
    nasm -f bin mbr.asm -o mbr.bin
    nasm -f bin bootloader.asm -o bootloader.bin
    dd if=mbr.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc
    dd if=bootloader.bin of=hd.img bs=512 count=5 seek=1 conv=notrunc
run:
    qemu-system-i386 -hda hd.img -serial null -parallel stdio
clean:
    rm -fr *.bin
```

我们根据上述指令，在 terminal 中首先输入

```
make build
```

即可将相应的.asm 文件转化为二进制文件并将其存入硬盘中。再输入

```
make run
```

即可使用 qemu 启动操作系统。

因为实验 1.1 的代码直接由参考资料中摘抄得来，故在实验报告中不再做多余的展示

## 实验 1.2

将 LBA28 读取硬盘的方式换成 CHS 读取，我们要修改 mbr.asm 的代码。

首先我们给出逻辑扇区号向 CHS 的转换公式

### LBA和CHS换算关系

类比十进制计数方法，可以给出从CHS计算LBA的公式：

$$\text{LBA} = (\text{S} + \text{H} * \text{BPB\_SecPerTrk} + \text{C} * \text{BPB\_NumHeads} * \text{BPB\_SecPerTrk}) - 1$$

进而推导出从LBA计算CHS的公式：

$$\text{S} = (\text{LBA} + 1) \% \text{BPB\_SecPerTrk}$$

$$\text{H} = (\text{LBA} + 1) / \text{BPB\_SecPerTrk} \% \text{BPB\_NumHeads}$$

$$\text{C} = (\text{LBA} + 1) / \text{BPB\_SecPerTrk} / \text{BPB\_NumHeads}$$

【LBA+1可以表示第几个扇区，因为LBA是从0开始编号的】

再由参考资料中所给的硬盘基本参数：

其中，关键参数如下。

参数	数值
驱动器号（DL寄存器）	80h
每磁道扇区数	63
每柱面磁头数（每柱面总的磁道数）	18

我们在程序中实现逻辑扇区号向 CHS 的转换公式的代码如下：

```

; 计算 S = (LBA + 1) % 63
xor dx, dx
inc ax      ; AX = LBA + 1
mov bx, 63
div bx      ; AX = (LBA + 1) // 63, DX = (LBA+1)%63
mov si, dx  ; SI = S

; 计算 H 和 C: AX = (LBA+1) // 63, DX = H (余数)
xor dx, dx
mov bx, 18
div bx      ; AX = C, DX = H
```

因为使用 CHS 模式需要使用中断，故我们将对应的数据存入对应的寄存器中，在使用 int 13h 中断，即可实现 CHS 模式的读取硬盘。修改后的代码保存在 mbr.1.2.asm 中

完整的 mbr1.2.asm 代码如下

```
org 0x7c00
[bits 16]
xor ax, ax ; eax = 0
; 初始化段寄存器, 段地址全部设为 0
mov ds, ax
mov ss, ax
mov es, ax
mov fs, ax
mov gs, ax

; 初始化栈指针
mov sp, 0x7c00
mov ax, 1 ; 逻辑扇区号第 0~15 位
mov cx, 0 ; 逻辑扇区号第 16~31 位
mov bx, 0x7e00 ; bootloader 的加载地址
call asm_read_hard_disk ; 读取硬盘
jmp 0x0000:0x7e00 ; 跳转到 bootloader

jmp $ ; 死循环

asm_read_hard_disk:
; 保存 LBA 号到 di
mov di, ax

; 计算 S = (LBA + 1) % 63
xor dx, dx
inc ax ; AX = LBA + 1
mov bx, 63
div bx ; AX = (LBA + 1) // 63, DX = (LBA+1)%63
mov si, dx ; SI = S

; 计算 H 和 C: AX = (LBA+1) // 63, DX = H (余数)
xor dx, dx
mov bx, 18
div bx ; AX = C, DX = H

; 设置 CHS 参数
mov cx, si
mov ch, al ; CH = 柱面低 8 位 (C ≤ 255)

; 设置 DH=磁头号, DL=80h
mov dh, dl ; DH = H
mov dl, 0x80 ; DL = 驱动器号

; 设置功能号及读取参数
mov ax, 0x0205 ; AH=02h 读扇区, AL=5 个扇区
```

```

    mov bx, 0x7e00

    int 0x13      ; 调用 BIOS 中断
    ret

times 510 - ($ - $$) db 0
db 0x55, 0xaa

```

通过执行前两个实验的 `bootloader.asm` 代码一致，故不做展示。

修改完 `mbr` 的内容后，在 `terminal` 中输入以下指令启动系统：

```

nasm -f bin bootloader.asm -o bootloader.bin

nasm -f bin mbr1.2.asm -o mbr1.2.bin

dd if=bootloader.bin of=hd.img bs=512 count=5 seek=1 conv=notrunc

dd if=mbr1.2.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc

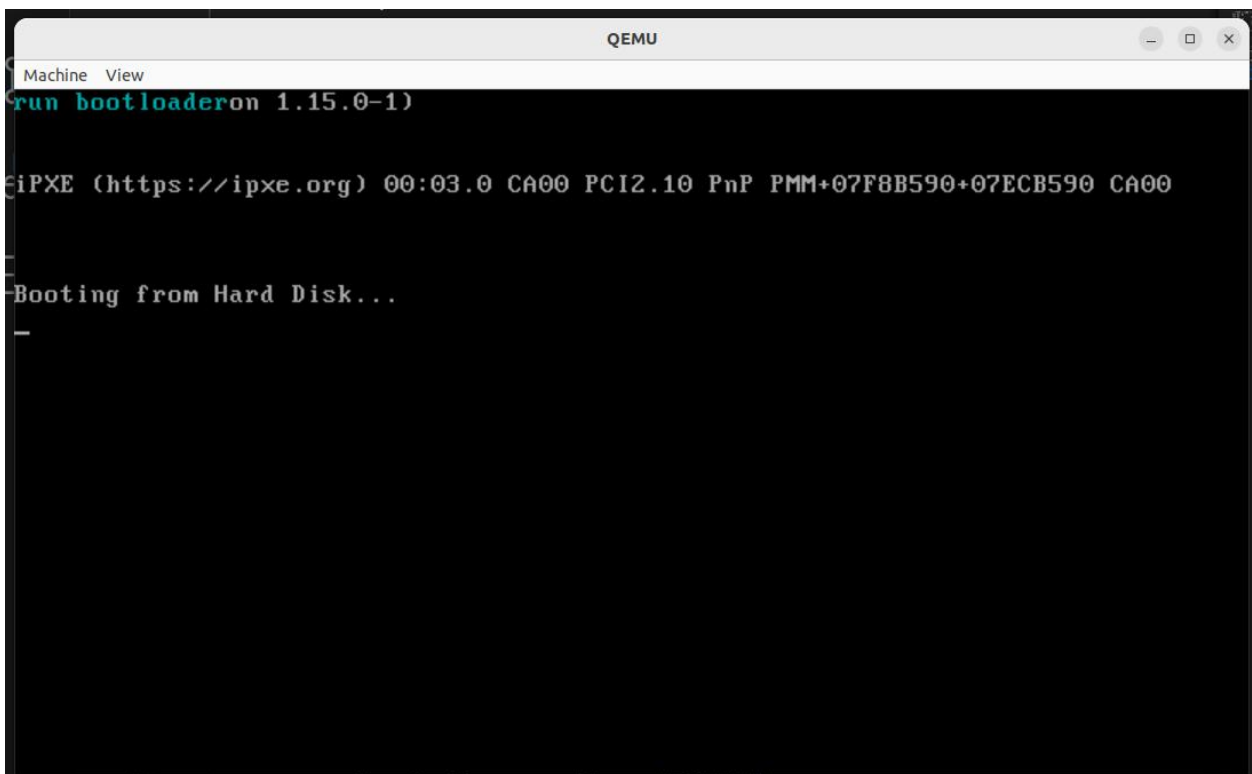
qemu-system-i386 -hda hd.img -serial null -parallel stdio

```

使用 `qemu` 启动系统后，可以在 `qemu` 显示屏上看到显示的字符串“`run bootloader`”

●实验结果展示：通过执行前述代码，可得下图结果。

因为两个实验虽然采取了不同的读取硬盘的方式，但 `bootloader.asm` 是一致的，故成功读取硬盘后在 `qemu` 显示屏上显示的是相同的结果，即一串字符串‘`run bootloader`’。结果如下图所示





## 实验任务 2

### ●任务要求:

#### Assignment 2

复现Example 2, 使用gdb或其他debug工具在进入保护模式的4个重要步骤上设置断点, 并结合代码、寄存器的内容等来分析这4个步骤, 最后附上结果截图。gdb的使用可以参考[appendix的“debug with gdb and qemu”部份](#)。

在这个lab的 `src` 文件夹中, 同样提供了makefile文件, 同学们可以自行学习通过 `make` 的方法, 更快捷地完成debug任务。

可以通过查看GDT的内容, 查看切换到保护模式的具体实现是怎麼样的? 在这个过程中, 你也可以思考一下, Linux-0.11的存储方式, 是小端存储还是大端存储呢?

### ●思路分析:

Example2 聚焦于实现进入保护模式。保护模式相对于实模式, 其内存地址增加到 32 位, 并且提供了段间的保护机制, 防止程序间随意访问地址带来的问题。所以要实现从实模式向保护模式的转换, 最重要的是实现段地址保护, 这需要我们给计算机段地址空间信息, 而此信息是通过段描述符(segment descriptor)来给出的。在保护模式下, 所有段描述符都会被集中放置, 这个集中放置的区域被称为全局描述符表(Global Descriptor Table, 简称 GDT), GDT 其实就是一个段描述符数组。

进入保护模式的步骤如下:

- 1、准备 GDT, 用 `lgdt` 指令加载 GDTR 信息。
- 2、打开第 21 根地址线。
- 3、开启 `cr0` 的保护模式标志位。
- 4、远跳转, 进入保护模式。

GDT 实际上是一个段描述符数组, 保存在内存中。GDT 的起始位置和大小由我们来确定, 保存在寄存器 GDTR 中。我们就是通过设置 GDTR 的内容来使 CPU 根据其内容找到 GDT 在内存中的位置, 然后从 GDT 中取出对应的段描述符。

剩下三步的操作相较简单并且有统一的流程操作, 只需使用相应代码实现即可。

### ●实验步骤:

Example2 中的代码已经在参考资料中给出, 实验 2 有关 `mbr.asm` 和 `bootloader.asm` 的代码直接从参考资料中下载得到, 故不再展示。

此部分主要展示使用 `gdb` 调试程序的结果, 在关键步骤访问相应寄存器, 查看进入保护模式的关键四个步骤的执行结果。

我们启动 `gdb` 的操作指令也可以通过创建 `Makefile` 文件来简化在 `terminal` 中操作指令的输入。为了更好的使用 `gdb` 来调试程序, 我们首先生成两个代码文件的 `symbol` 文件, 即符号表。符号表会为 `gdb` 提供源代码和标识符等 `debug` 信息。

在 Linux 下, 符号表是保存在 ELF 格式的文件下的。注意到我们之前使用 `nasm` 编译生成的是 `bin` 格式文件, `bin` 文件仅包含代码和数据, 而 ELF 格式文件除了包含 `bin` 文件的内容外, 还包含有其他



信息，例如符号表等。因此，我们如果想给 gdb 提供 debug 所需的符号表，我们就不能一步地从汇编代码生成 bin 格式文件，而是需要先编译汇编代码生成一个可重定位文件，这个文件也是 ELF 格式的。然后我们使用这个可重定位文件生成可执行文件和 bin 格式文件，这个可执行文件就是我们给 gdb 提供的符号表所在的 ELF 文件。

接下来，我们就来生成符号表。

我们首先删除 mbr.asm 和 bootloader.asm 的 org 语句，因为我们会在链接的过程中指定他们代码和数据的起始地址，其效果和 org 指令完全相同。

我们编译 mbr.asm，生成可重定位文件 mbr.o。其中，-g 参数是为了加上 debug 信息。

```
nasm -o mbr.o -g -f elf32 mbr.asm
```

然后我们为可重定位文件 mbr.o 指定起始地址 0x7c00，分别链接生成可执行文件 mbr.symbol 和 mbr.bin

```
ld -o mbr.symbol -melf_i386 -N mbr.o -Ttext 0x7c00
```

```
ld -o mbr.bin -melf_i386 -N mbr.o -Ttext 0x7c00 --oformat binary
```

对于 bootloader.asm，我们执行上述类似的操作。

```
nasm -o bootloader.o -g -f elf32 bootloader.asm
```

```
ld -o bootloader.symbol -melf_i386 -N bootloader.o -Ttext 0x7e00
```

```
ld -o bootloader.bin -melf_i386 -N bootloader.o -Ttext 0x7e00 --oformat binary
```

然后将 mbr.bin 和 bootloader.bin 分别写入 hd.img，写入的位置是 lab2-Example 2 中指定的位置。

```
dd if=mbr.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc
```

```
dd if=bootloader.bin of=hd.img bs=512 count=5 seek=1 conv=notrunc
```

因此，我们在 makefile 文件中有关 make symbol 和 make debug 部分的代码如下：

```
symbol:
```

```
@nasm -g -f elf32 mbr.asm -o mbr.o
```

```
@${LD} -o mbr.symbol -melf_i386 -N mbr.o -Ttext 0x7c00
```

```
@nasm -g -f elf32 bootloader.asm -o bootloader.o
```

```
@${LD} -o bootloader.symbol -melf_i386 -N bootloader.o -Ttext 0x7e00
```

```
debug:
```

```
@qemu-system-i386 -s -S -hda hd.img -serial null -parallel stdio &
```

```
@sleep 1
```

```
@gnome-terminal -- gdb -q -x gdbinit
```

gdb 的命令也可以预先写入到文件中，在启动 gdb 后自动加载执行，例如我们把 gdb 的初始化命令写入文件 gdbinit 中，如下所示

```
target remote:1234
```

```
set disassembly-flavor intel
```

```
add-symbol-file mbr.symbol 0x7c00
```

```
add-symbol-file bootloader.symbol 0x7e00
```

我们在 terminal 中进入对应文件目录下输入以下指令：

```
make symbol
make debug
```

即可进入 gdb 的 debug 界面对程序进行调试。

进入 gdb 后，我们在 MBR 的第一条指令处设置断点。

```
b *0x7c00
```

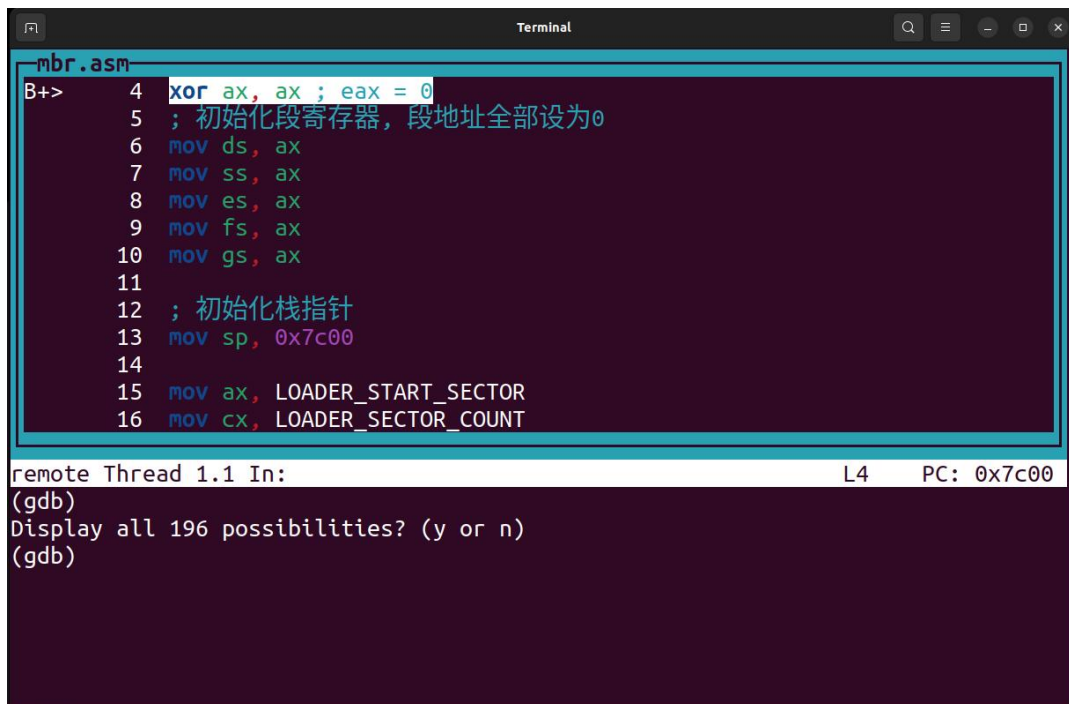
输入 c 执行到断点 0x7c00 处暂停。

```
c
```

然后我们打开可以显示源代码的窗口。

```
layout src
```

结果如下：



```
mbr.asm
B+> 4 xor ax, ax ; eax = 0
      5 ; 初始化段寄存器，段地址全部设为0
      6 mov ds, ax
      7 mov ss, ax
      8 mov es, ax
      9 mov fs, ax
     10 mov gs, ax
     11
     12 ; 初始化栈指针
     13 mov sp, 0x7c00
     14
     15 mov ax, LOADER_START_SECTOR
     16 mov cx, LOADER_SECTOR_COUNT

remote Thread 1.1 In: L4 PC: 0x7c00
(gdb)
Display all 196 possibilities? (y or n)
(gdb)
```

随后我们在进入保护模式的 4 个重要步骤上设置断点

首先我们来到 bootloader 的设置 gdt 内容的地方，在此设置断点：

```
b 0x7e00
```

```
c
```

```
b 18
```

```
c
```

结果如图：

```

bootloader.asm
13  inc esi
14  add ebx,2
15  loop output_bootloader_tag
16
17  ;空描述符
B+> 18  mov dword [GDT_START_ADDRESS+0x00],0x00
19  mov dword [GDT_START_ADDRESS+0x04],0x00
20
21  ;创建描述符,这是一个数据段,对应0~4GB的线性地址空间
22  mov dword [GDT_START_ADDRESS+0x08],0x0000ffff ; 基地址为0,段界限为0xF
23  mov dword [GDT_START_ADDRESS+0x0c],0x00cf9200 ; 粒度为4KB,存储器段描述
24
25  ;建立保护模式下的堆栈段描述符
26  mov dword [GDT_START_ADDRESS+0x10],0x00000000 ; 基地址为0x00000000,界
27  mov dword [GDT_START_ADDRESS+0x14],0x00409600 ; 粒度为1个字节
28
29  ;建立保护模式下的显存描述符
30  mov dword [GDT_START_ADDRESS+0x18],0x80007fff ; 基地址为0x000B8000,界

remote Thread 1.1 In: output_bootloader_tag L18 PC: 0x7e24
(gdb) n
(gdb) b 18
Breakpoint 3 at 0x7e24: file bootloader.asm, line 18.
(gdb) c
Continuing.

Breakpoint 3, output_bootloader_tag () at bootloader.asm:18
(gdb)

```

我们继续执行程序，使 GDT 设置完成，输入以下指令，查看 GDT 的 5 个段描述符的内容。

x/5xg 0x8800

```

bootloader.asm
30  mov dword [GDT_START_ADDRESS+0x18],0x80007fff ; 基地址为0x000B8000,界
31  mov dword [GDT_START_ADDRESS+0x1c],0x0040920b ; 粒度为字节
32
33  ;创建保护模式下平坦模式代码段描述符
34  mov dword [GDT_START_ADDRESS+0x20],0x0000ffff ; 基地址为0,段界限为0xF
35  mov dword [GDT_START_ADDRESS+0x24],0x00cf9800 ; 粒度为4kb,代码段描述符
36
37  ;初始化描述符表寄存器GDTR
38  mov word [pgdt], 39 ;描述符表的界限
39  lgdt [pgdt]
40
> 41  in al,0x92 ;南桥芯片内的端口
42  or al,0000_0010B
43  out 0x92,al ;打开A20
44
45  cli ;中断机制尚未工作
46  mov eax,cr0
47  or eax,1

remote Thread 1.1 In: output_bootloader_tag L41 PC: 0x7e89
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) x/5xg 0x8800
0x8800: 0x0000000000000000 0x00cf92000000ffff
0x8810: 0x0040960000000000 0x0040920b80007fff
0x8820: 0x00cf98000000ffff
(gdb)

```

可以看到 GDT 的内容和我们的设置相吻合。

随后我们再继续执行到其他几个步骤的位置，查看相关的变化

打开第 21 根地址线，设置 PE 位，和进行远跳转进入保护模式

```
bootloader.asm
39 lgdt [pgdt]
40
41 in al,0x92 ;南桥芯片内的端口
42 or al,0000_0010B
43 out 0x92,al ;打开A20
44
45 cli ;中断机制尚未工作
46 mov eax,cr0
47 or eax,1
48 mov cr0,eax ;设置PE位
49
50 ;以下进入保护模式
> 51 jmp dword CODE_SELECTOR:protect_mode_begin
52
53 ;16位的描述符选择子：32位偏移
54 ;清流水线并串行化处理器
55 [bits 32]
56 protect_mode_begin:

remote Thread 1.1 In: output_bootloader_tag L51 PC: 0x7e9a
0x8810: 0x0040960000000000 0x0040920b80007fff
0x8820: 0x00cf98000000ffff
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
```

进入保护模式。然后将选择子放入对应的段寄存器。

使用指令

```
info register
```

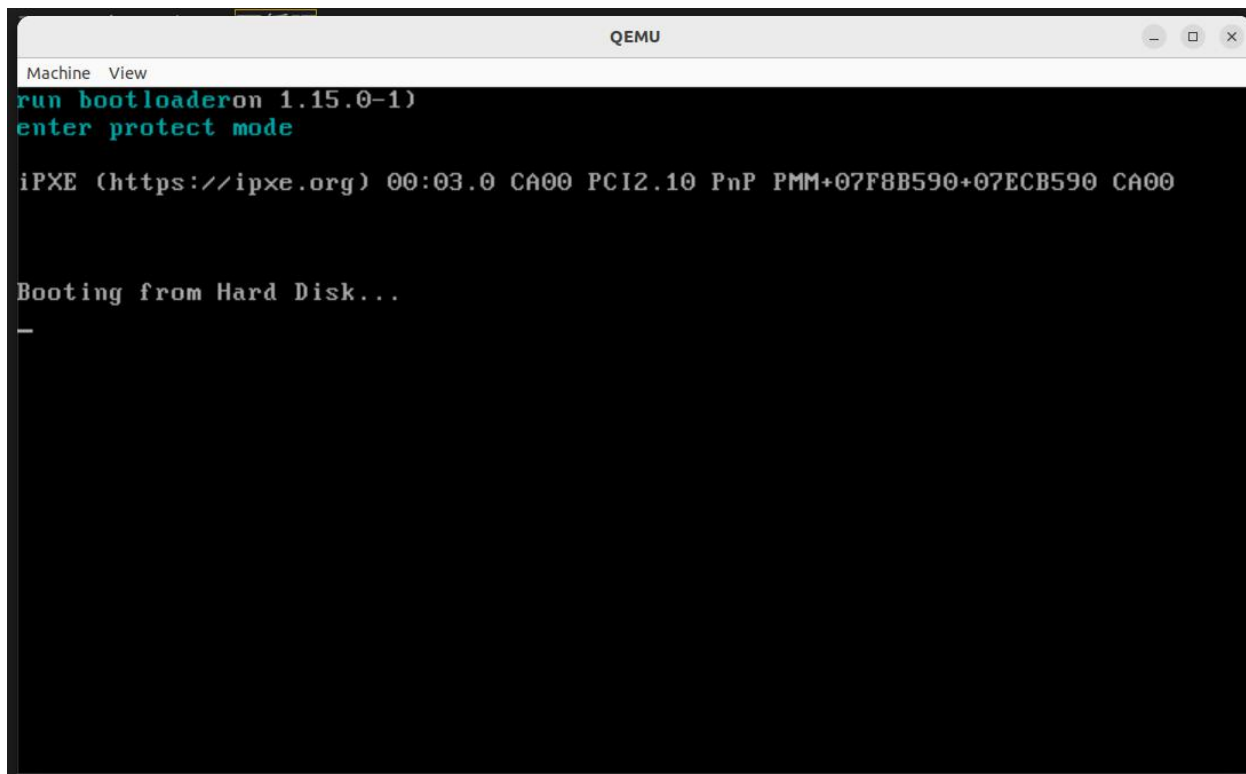
```
bootloader.asm
52
53 ;16位的描述符选择子：32位偏移
54 ;清流水线并串行化处理器
55 [bits 32]
56 protect_mode_begin:
57
58 mov eax, DATA_SELECTOR ;加载数据段(0..4GB)选择子
59 mov ds, eax
60 mov es, eax
61 mov eax, STACK_SELECTOR
62 mov ss, eax
63 mov eax, VIDEO_SELECTOR
64 mov gs, eax
65
> 66 mov ecx, protect_mode_tag_end - protect_mode_tag
67 mov ebx, 80 * 2
68 mov esi, protect_mode_tag
69 mov ah, 0x3

remote Thread 1.1 In: protect_mode_begin L66 PC: 0x7eb9
[ IOPL=0 PF ]
cs      0x20      32
ss      0x10      16
ds      0x8       8
es      0x8       8
fs      0x0       0
gs      0x18      24
fs_base 0x0       0
gs_base 0xb8000   753664
--Type <RET> for more, q to quit, c to continue without paging--
```

可以看到选择子正确的放入对应的段寄存器中。

- 实验结果展示：通过执行前述代码，可得下图结果。

调试结果图片在实验步骤中已经展现，下面是整个程序正确运行的结果：



可以看到在 qemu 显示屏上正确显示了 ‘run bootloader’ 和 ‘enter protect mode’ 两个字符串

## ----- 实验任务 3 -----

- 任务要求：

### Assignment 3

改造“Lab2-Assignment 4”为32位代码，即在加载到保护模式后执行自定义的汇编程序。

提示：①在lab2中使用中断来实现功能的同学，现在会出错，为什么？怎么解决？②在lab2中直接将字符输出到qemu显存的同学，如果直接复制粘贴代码，同样也会出错，为什么？怎么解决？

- 思路分析：

实验 3 的任务为改造 “Lab2-Assignment 4” 为 32 位代码，在加载到保护模式后执行。注意在保护模式中，将不能在使用实模式下的中断，并且要注意相应段地址的设置，因此在实现字符的显示时要注意段地址的设置并找到正确的段地址来存放相应的数据。

还要注意保护模式下内存地址扩展到了 32 位，因此对于 lab2 在实模式下运行的程序中寄存器的使用还要做出相应的修改。

## ●实验步骤：

我们直接在 lab2 assignment4 代码的基础上进行修改，因为我们在 lab2 中利用了实模式中断来实现延迟操作，因此在保护模式下要将其换成一个进行有限次数空操作循环的函数，来实现延迟操作，此延迟函数的代码如下：

```
delay:
    pusha
    mov ecx, 0x1FFFFFF
    delay_loop:
    dec ecx
    jnz delay_loop
    popa
    ret
```

还要注意，在 lab2 中我们是通过直接在 qemu 显示屏的显存地址存放数据来实现显示字符串操作的，但是在保护模式下，由于我们已经对应的设置了各个部分的段地址，因此我们输出字符要使用 gs 段寄存器来设置数据来实现打印字符的操作。

修改后的 draw\_char 模块代码如下所示：

```
draw_char:
    pusha
    xor ax, ax
    mov al, [row]
    mov bx, 80
    mul bx
    add al, [col]
    adc ah, 0
    shl ax, 1
    mov di, ax

    mov ebx, [charIndex]
    mov esi, chars
    mov al, [esi + ebx]

    mov ah, [color]
    mov [gs:di], ax ; 使用 gs 段寄存器来储存数据
    popa
    ret
```

除了这些地方的代码修改外，我还将 draw\_id\_name 处的代码进行了相应修改，仿造资料中打印字符串的代码，修改后的代码更加简洁，修改后的结果如下：







## Section 3 实验总结与心得体会

1、通过本次实验，我对如何通过 MBR 使用 LBA28 或 CHS 模式读取硬盘有了全新的认识，并且学会了 LBA28 和 CHS 两种读取硬盘模式的切换。对操作系统读取硬盘的步骤有了更深的理解，学会了通过读取硬盘来加载实现更大内存的程序。

2、我对保护模式的功能有了全新的理解，对保护模式相较于实模式增加的功能与用途有了全新的认识。知道了如何设置段描述符，并通过设置段描述符等一系列操作实现从实模式到保护模式的切换。

3、通过对保护模式更深层次的理解，我学会了在保护模式下运行自己编写的程序，对保护模式的认识更加深了一层。

4、通过对相关实验资料中对 debug 部分的讲解，我对使用 gdb 调试代码程序有了更深的理解，知道了如何生成 symbol 文件来调用 src 以实现调试过程中执行代码的显示。

5、通过 Makefile 文件的编写，我对如何快速实现重复指令的使用有了清晰的认识，学会了如何通过编写 Makefile 文件来简化程序指令操作。

## Section 4 对实验的改进建议和意见

希望对于保护模式下程序编写需要注意的事项有相应的讲解。

## Section 5 附录：参考资料清单

[lab3 • apshuang/SYSU-2025-Spring-Operating-System - 码云 - 开源中国](#)

[appendix/debug with gdb and qemu/README.md • apshuang/SYSU-2025-Spring-Operating-System - Gitee.com](#)

<https://zhuanlan.zhihu.com/p/608292324>

<https://blog.csdn.net/brainkick/article/details/7583727>