

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。

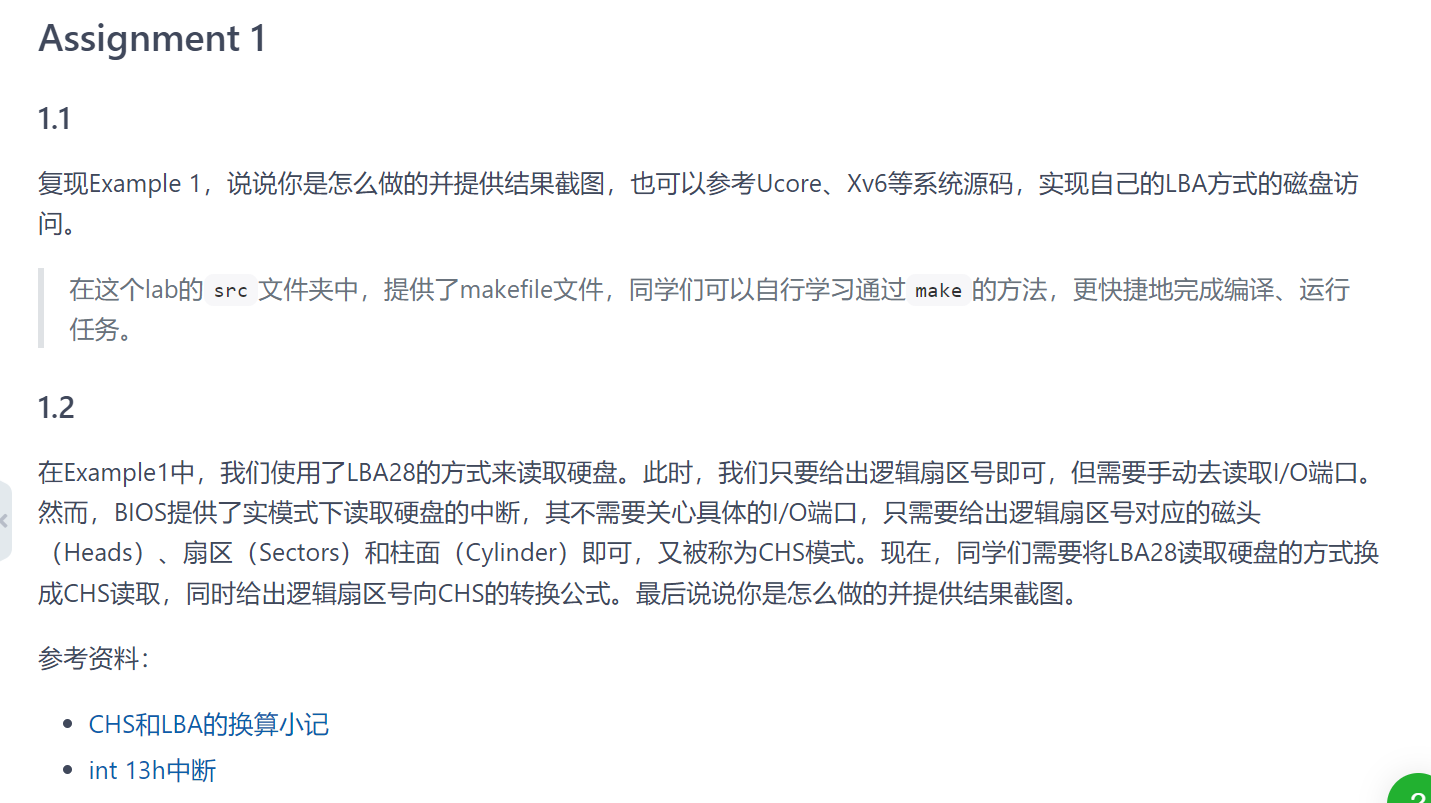
**实验任务3：**

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：



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：



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



再由参考资料中所给的硬盘基本参数：



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boatloader.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

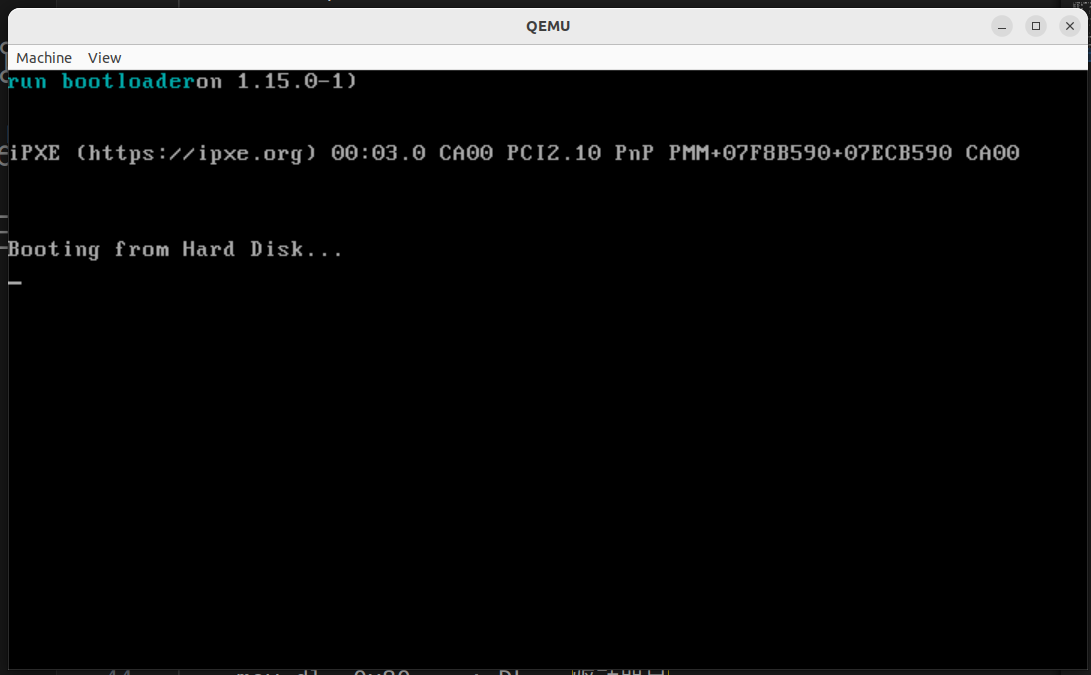
dd if=mbr1.2.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc

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”

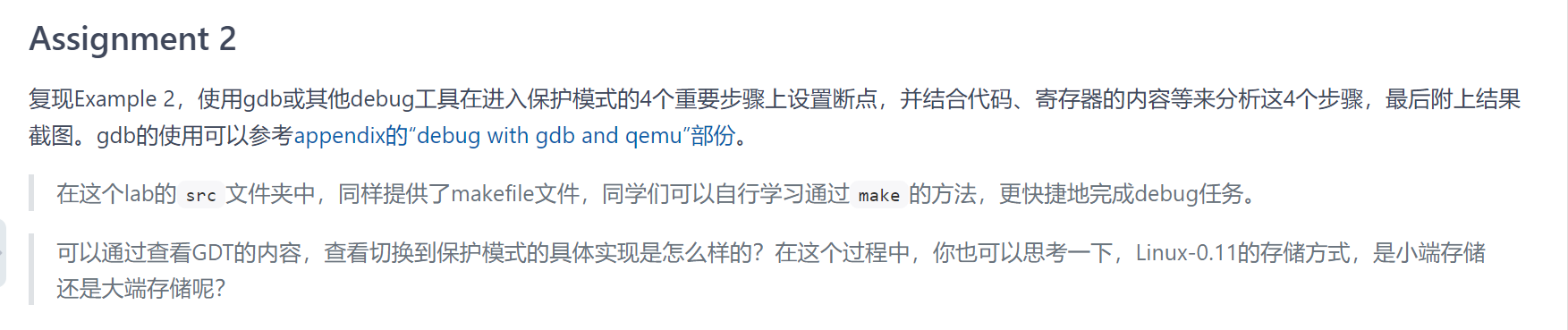
实验结果展示：通过执行前述代码，可得下图结果。

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：



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 0x7c00

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

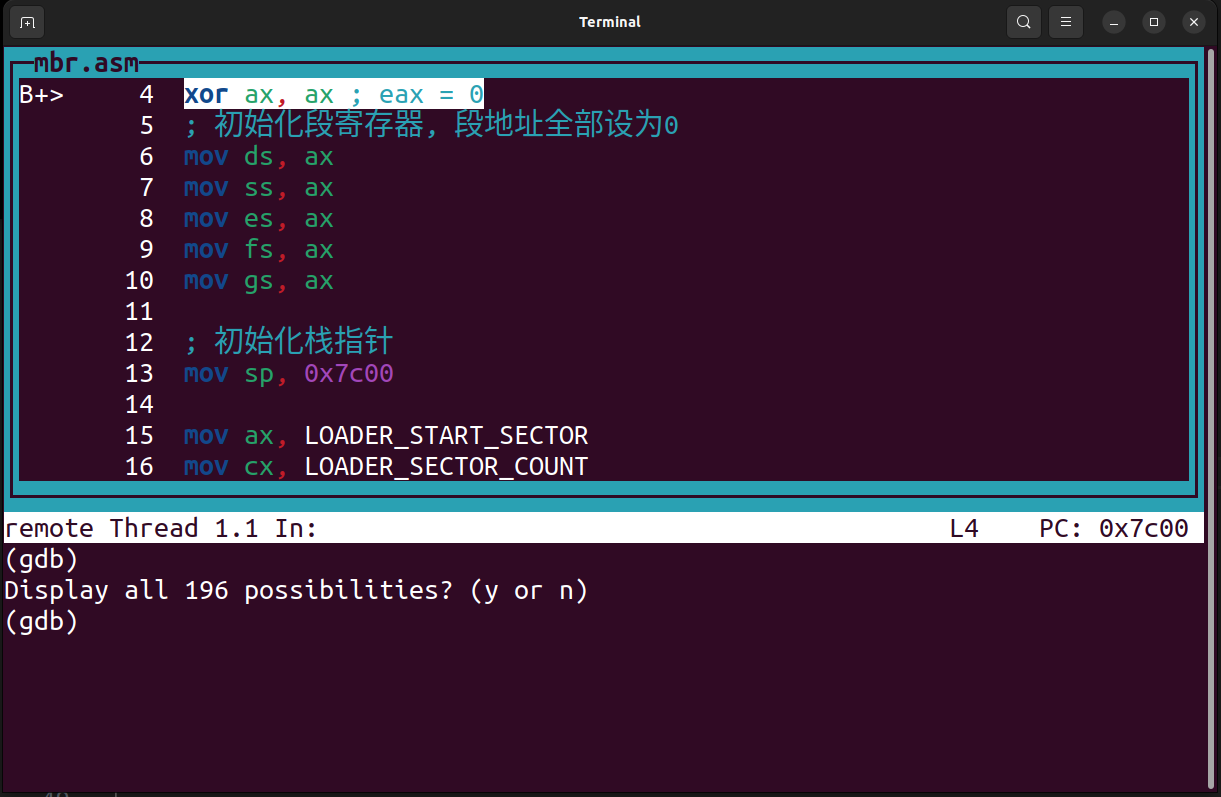
输入c执行到断点0x7c00处暂停。

c

然后我们打开可以显示源代码的窗口。

layout src

结果如下：



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：

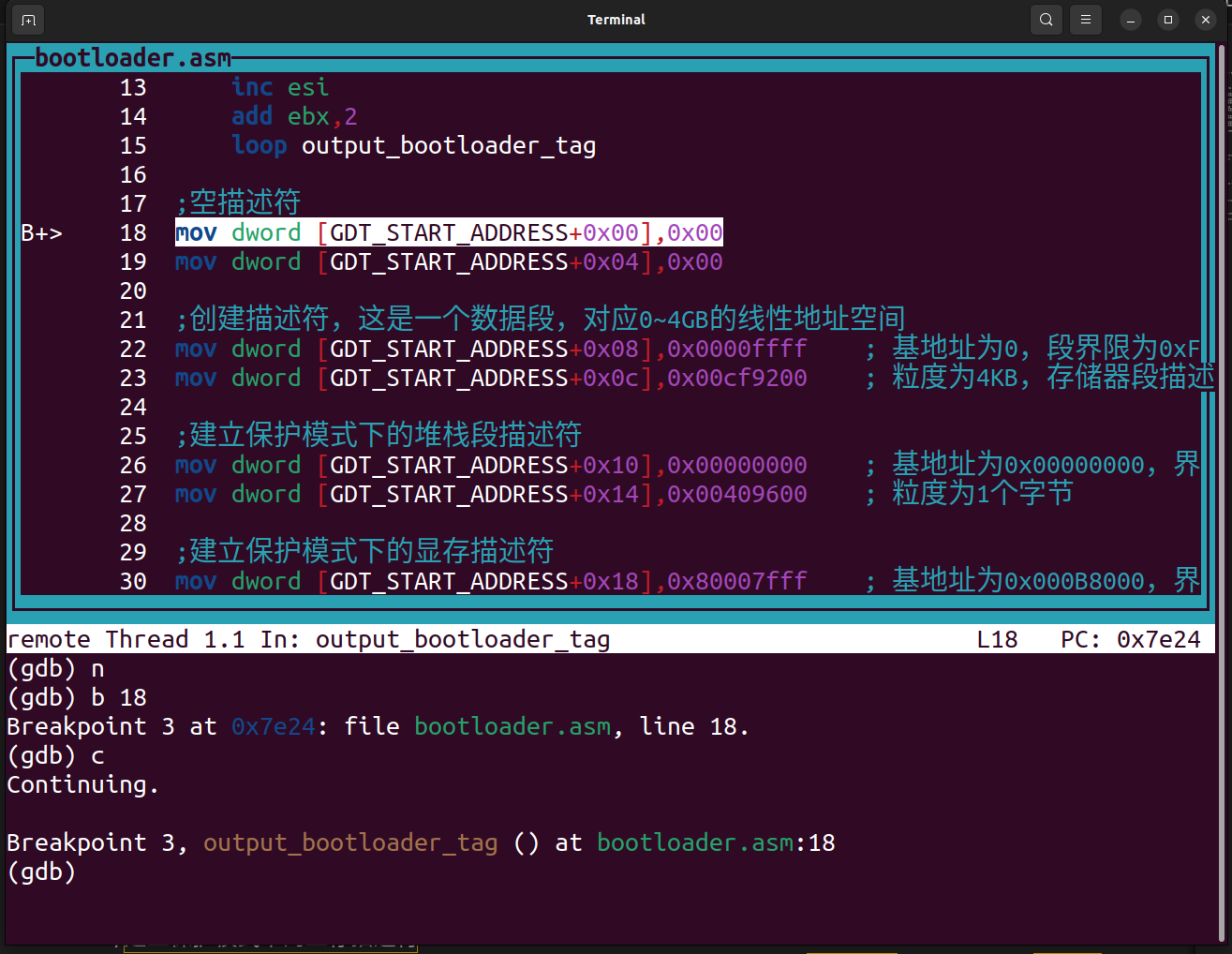
b 0x7e00

c

b 18

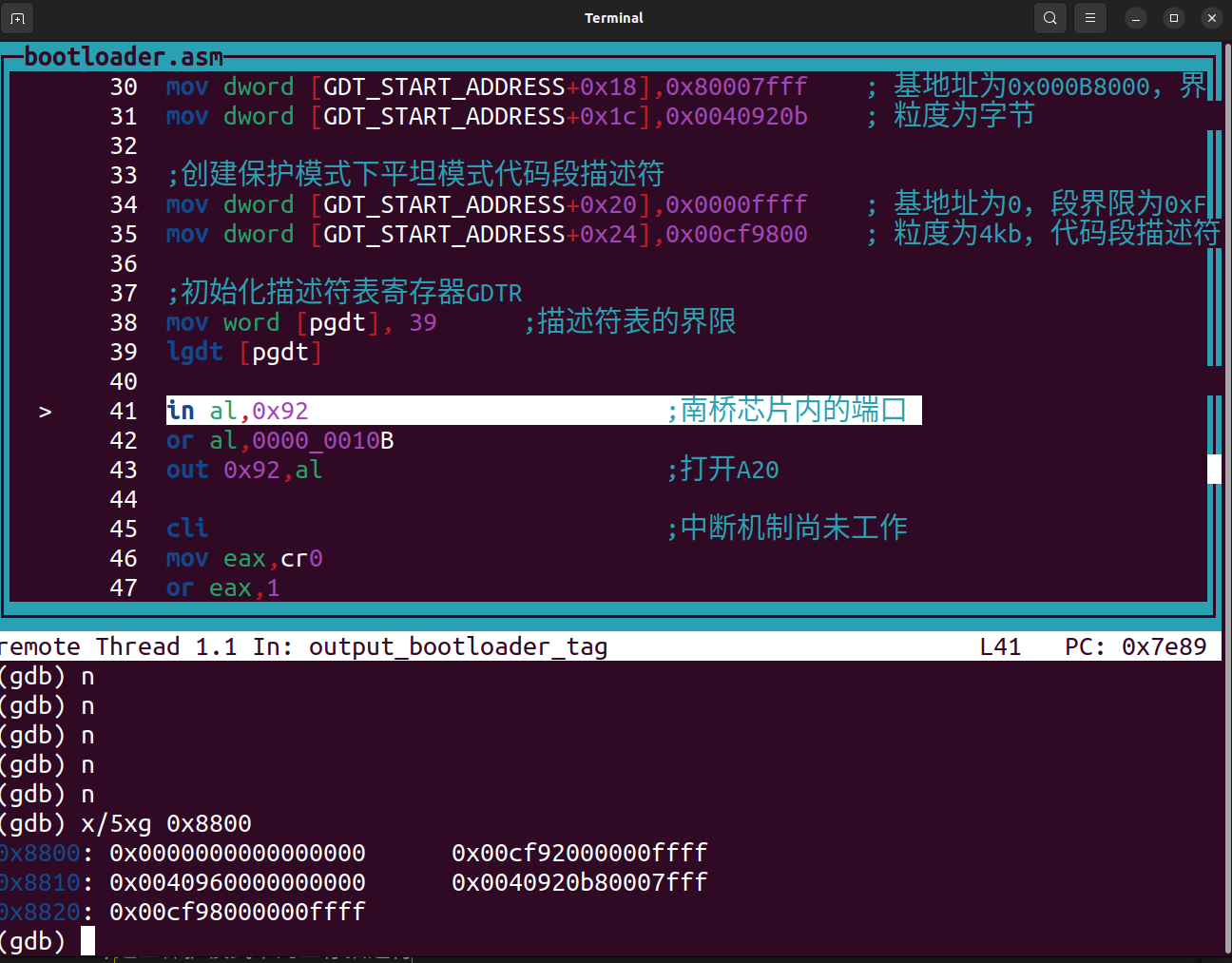
c

结果如图：



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。

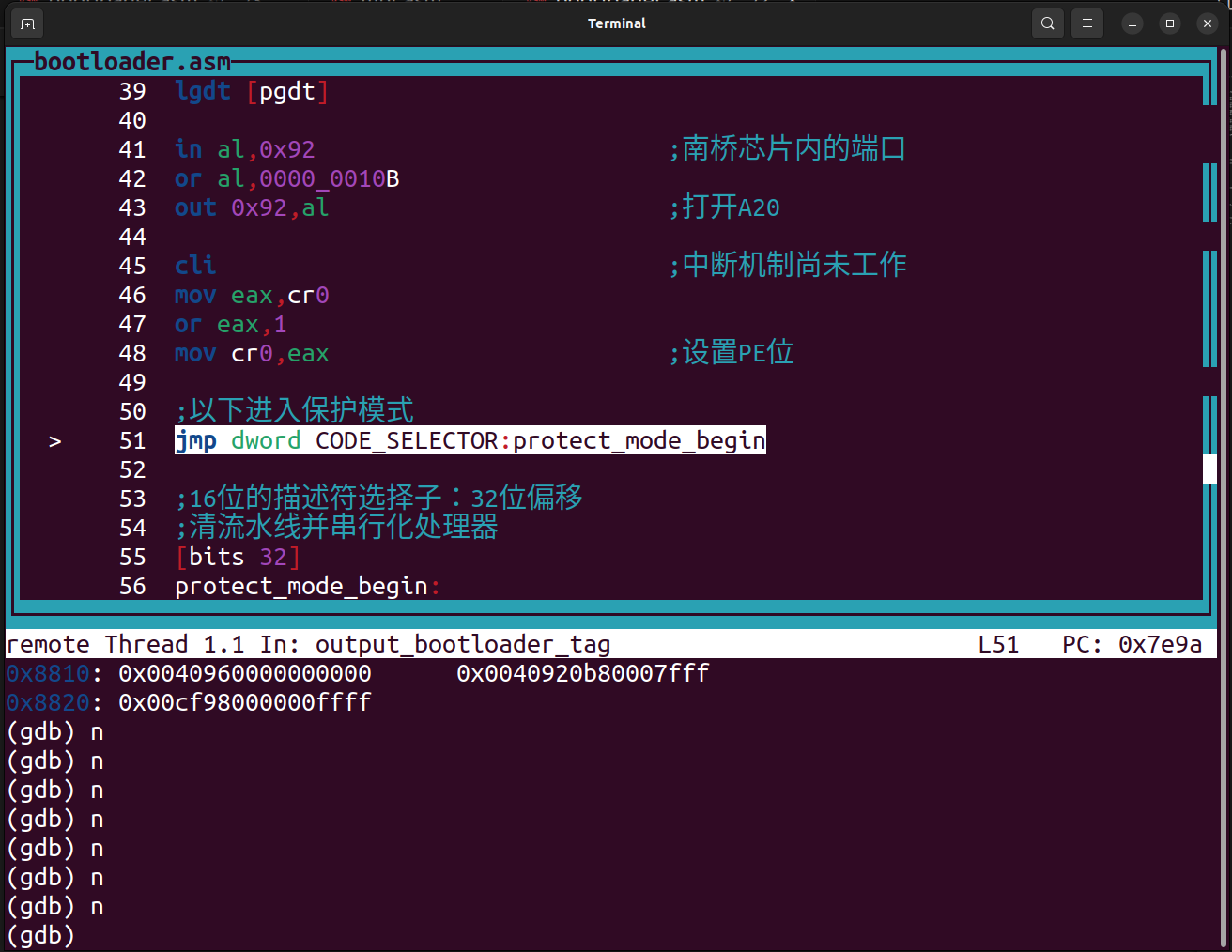
x/5xg 0x8800



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

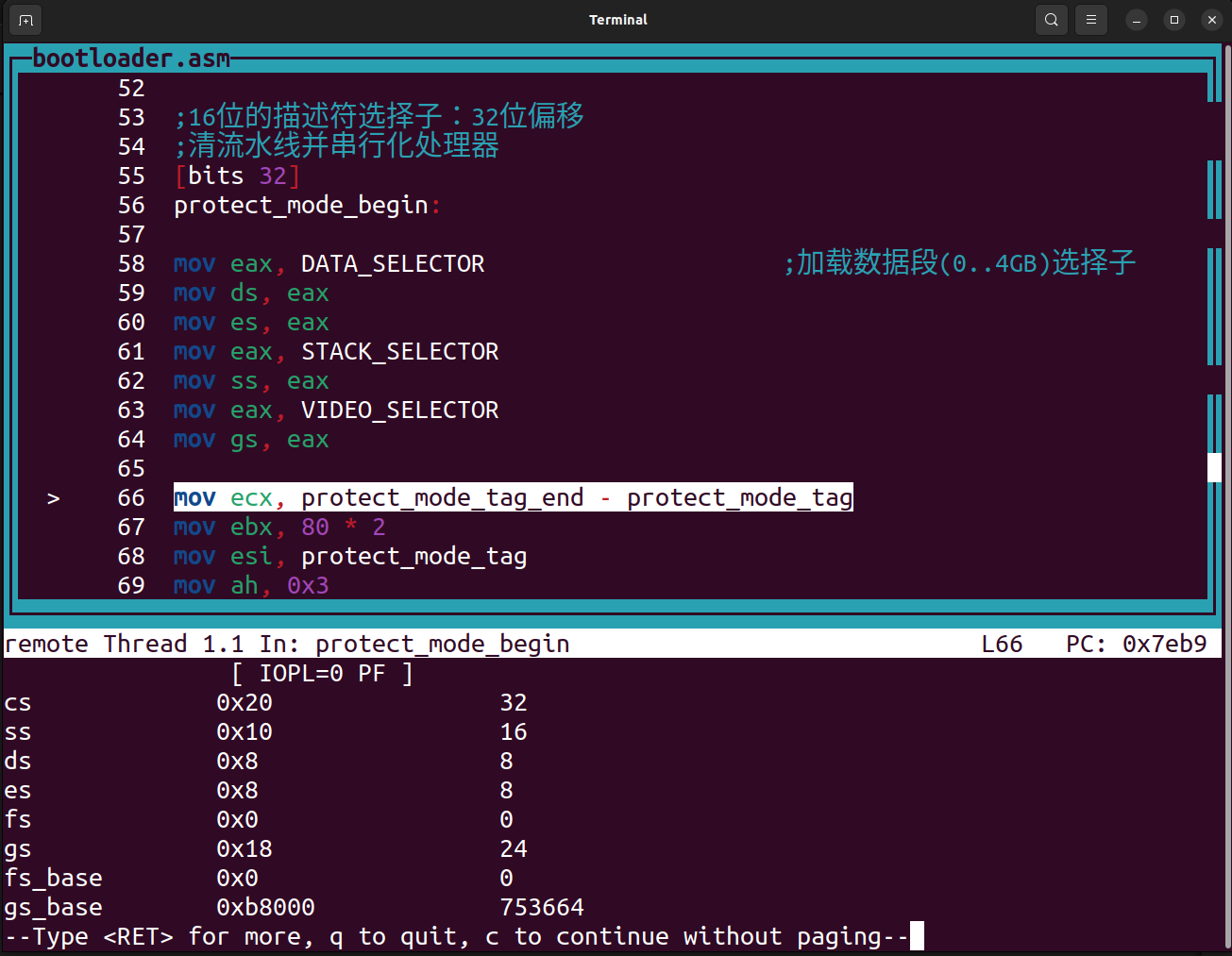
打开第21根地址线，设置PE位，和进行远跳转进入保护模式



进入保护模式。然后我们将选择子放入对应的段寄存器。

使用指令

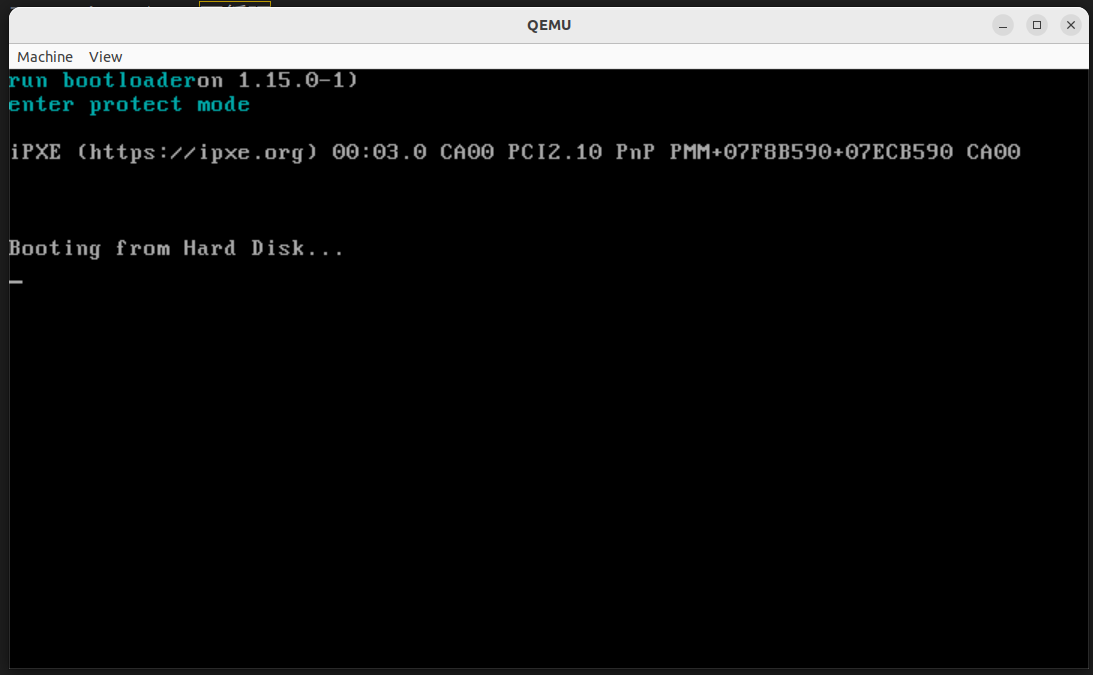
info register



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。

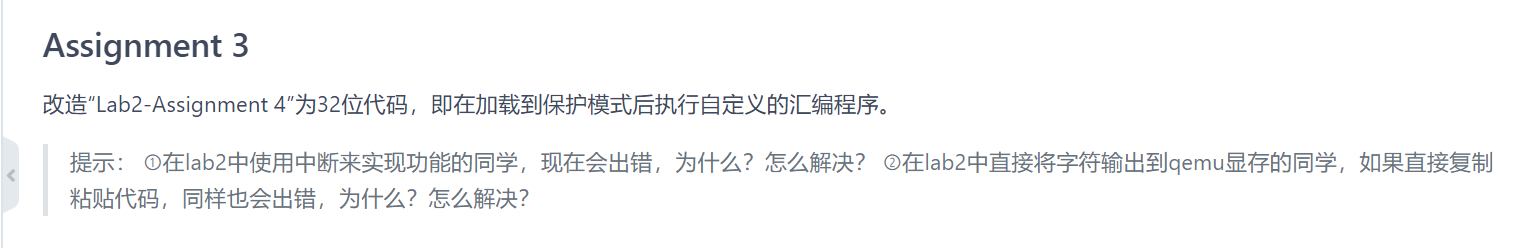
调试结果图片在实验步骤中已经展现，下面是整个程序正确运行的结果：



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：



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, 0x1FFFFF

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：

draw\_id\_name:

pusha

mov ecx, name\_tag\_end - name\_tag

mov ebx, 995 \* 2

mov esi, name\_tag

mov ah, 0x74

output\_name\_tag:

mov al, [esi]

mov word[gs:ebx], ax

add ebx, 2

inc esi

loop output\_name\_tag

popa

ret

name\_tag db '23336342-ZWX'

name\_tag\_end:

在一些有关寄存器的设置方面的细节处理之后，我们便可以在保护模式下运行此字符串回旋显示程序。程序代码直接在bootloader.asm下附接即可。

同实验2一样创建的Makefile文件，我们直接在terminal中使用make build 和 make run指令即可运行程序。

实验结果展示：通过执行前述代码，可得下图结果。

运行后程序如图所示：



**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](https://gitee.com/apshuang/sysu-2025-spring-operating-system/tree/master/lab3)

[appendix/debug\_with\_gdb\_and\_qemu/README.md · apshuang/SYSU-2025-Spring-Operating-System - Gitee.com](https://gitee.com/apshuang/sysu-2025-spring-operating-system/blob/master/appendix/debug_with_gdb_and_qemu/README.md)

[https://zhuanlan.zhihu.com/p/608292324](https://zhuanlan.zhihu.com/p/608292324" \o "1)

[https://blog.csdn.net/brainkick/article/details/7583727](https://zhuanlan.zhihu.com/p/608292324" \o "1)

÷