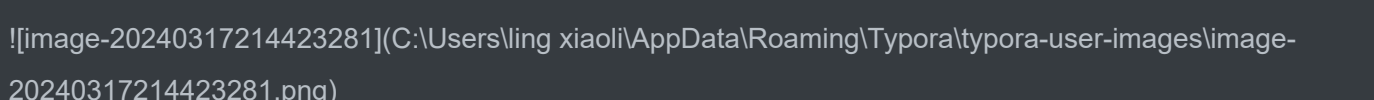
```
; 设置代码段的起始位置
BITS 16
SECTION MBR vstart=0x7c00

; 初始化堆栈
mov ax, cs
mov ds, ax
mov es, ax
mov ss, ax
mov sp, 0x7c00


_start:
    ; 清屏
    mov ax, 0600h
    mov bx, 0700h
    mov cx, 0
    mov dx, 184fh
    int 10h

    ; 打印菜单
    mov si, menu_windows
    call print_string
    mov si, menu_linux
    call print_string


    ; 处理用户输入
    call handle_input

    ; 无效选择，重新显示菜单
    jmp _start


    ; 处理用户输入
handle_input:
    mov ah, 0
    int 16h
    cmp ah, 0
    je handle_input

    cmp al, '1'
    je windows_selected
    cmp al, '2'
    je linux_selected
    jmp handle_input


; 用户选择 Windows 分区的操作
windows_selected:
    ;把MBR从0x7c00移到0x600
```

```
    xor ax, ax
    mov ss, ax
    mov sp, 0x7c00
    mov es, ax
    mov ds, ax
    mov si, 0x7c00
    mov di, 0x600
    mov cx, 0x200
    cld
    rep movsb

    ;这步是大坑之一，我们要跳转到新地址继续执行MBR剩余的代码，0x659需要你先编译再反编译看看sti这条命令的地址，注意是0x600加上偏移（即第几个字节）
    push ax
    push word 0x659
    retf

    ;这里有点取巧，直接跳过了搜索活动分区的步骤，因为我们知道Windows的加载器就在第一个分区的活动扇区，0x7be就是分区表第一个分区的入口（是移动到0x600后的位置，这里应该是固定的），下面是采用传统的int 13h 用CHS的方法，也可以用LBA
    sti
    mov bp, 0x7be
    mov dl,[bp+0x0]
    mov dh,[bp+0x1]
    mov cl,[bp+0x2]
    mov ch,[bp+0x3]
    mov bx, 0x7c00
    mov ax, 0x0201
    int 0x13
    jmp 0x0000:0x7c00

; 用户选择 Linux 分区的操作
linux_selected:
    mov si, message_linux
    call print_string
    ; 建栈
    cli
    nop
    nop
    xor ax, ax
    mov ds, ax
    mov ss, ax
    mov sp, 0x2000
    sti
;未知功能
    mov dl, 0x80
    push dx
    mov bx,0x55aa

;使用LBA模式读取start程序，读取到内存0x7000处
    mov si,0x6000
    xor ax,ax
    mov [si+0x4],ax
    inc ax
    mov [si-0x1],al
```

```
    mov [si+0x2],ax
    mov word [si],0x10
    mov ebx,0x00000001
    mov [si+0x8],ebx
    mov ebx,0x00000000
    mov [si+0xc],ebx
    mov word [si+0x6],0x7000
    mov ah,0x42
    int 0x13

;将start程序从0x7000移动到指定的启始地址位置，在这里是0x8000，并跳转到start程序
    pusha
    push ds
    mov bx,0x7000
    mov cx,0x100
    mov ds,bx
    xor si,si
    mov di,0x8000
    mov es,si
    cld
    rep movsw
    pop ds
    jmp 0x8000



; 打印字符串
print_string:
    lodsb
    or al, al
    jz .done_printing
    mov ah, 0x0E
    int 10h
    jmp print_string
.done_printing:
    ret


; 定义启动分区选项
menu_windows db "1. Windows ", 0
menu_linux   db "2. Linux ", 0

; 初始化选中分区
selected_partition db 0
; Windows 分区选项
message_windows db "Windows", 0

; Linux 分区选项
message_linux   db "Linux", 0


; MBR 结束标志
times 510-($-$$) db 0
```

```
dw 0xAA55
```

在修改硬盘文件以前，记得先保存原本硬盘的bin文件，以上编译出bin文件后，再到vscode利用hex editor，把原硬盘的从441开始到最后的字节拷贝过来修改（坑2：一定不要忘记拷贝磁盘签名那四个字节）

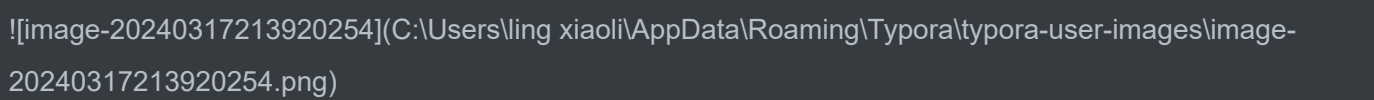


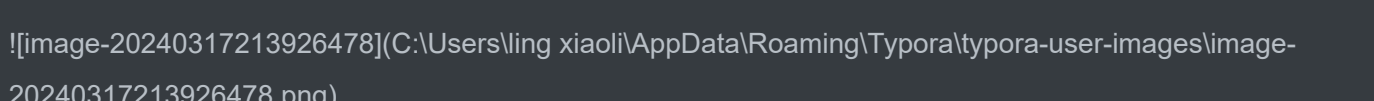
然后OK了；相关命令如下：

编译新bin：nasm -f bin -o new.bin mbr.asm 替换新bin：dd if=new.bin of=mydisk7.raw bs=512 count=1 复原原来的bin：dd if=mbr.bin of=mydisk7.raw bs=512 count=1 打开：qemu-system-x86_64 -bios D:\QEMU\share\bios.bin -drive file=mydisk7.raw,format=raw -m 5G -smp 8

附加一些代码的注释，从网上搜到的，以及我自己的一些注释：

Windows：





Linux（这个是我根据网上资料写的Linux的注释，不一定全对）

```
class One。启动磁盘的检查以及载入start程序前的准备

00000000  EB63              jmp short 0x65  //无条件跳转到偏移地址 0x65 处
00000065  FA                cli //CPU执行cli指令，这是禁止CPU中断发生，确保当前运行的代码不会被打断。
00000066  90                nop
00000067  90                nop  //CPU执行nop指令，这是个空操作指令，只是用于保证上下指令之间增加一个稳定时间。
00000068  F6C280            test dl,0x80 //这里测试dl寄存器是不是80开头的，即测试是否是硬盘驱动
0000006B  7405              jz 0x72 //假如上一条指令dl为80开头，那么代表是硬盘驱动，如果不是则跳到72
00000072  B280              mov dl,0x80 //判断dl寄存器是否80开头（代表硬盘驱动器），如果不是，则直接将0x80送入dl覆盖，确保在后续操作中正确地识别和操作硬盘驱动器
00000074  EA797C0000        jmp 0x0:0x7c79 //ljmp 到下一条指令，因为一些虚假的 BIOS 会跳转到 07C0:0000 而不是 0000:7C00。为了以防万一，用此命令进行纠正
00000079  31C0              xor ax,ax //将AX寄存器的值，与自身做逻辑异或计算，始终得到0x0，并将结果送入AX寄存器
```

0000007B 8ED8 mov ds,ax //此时AX=0x0，将AX的值设置到DS寄存器中，最终DS=0x0

0000007D 8ED0 mov ss,ax //此时AX=0x0，将AX的值设置到SS寄存器中，最终SS=0x0

0000007F BC0020 mov sp,0x2000 //将SP寄存器的值设为SP=0x2000，SS 和 SP 两个寄存器都有了明确的值，代表此段代码执行到这里，正式构建了一个栈空间SS:SP=00:2000

00000082 FB sti //执行指令sti。 允许CPU中断发生。这条指令和上面00000065 FA cli指令形成配合，在cli和sti包围中的代码不会被外部中断，以确保它们能够一次正确运行。

00000083 A0647C mov al,[0x7c64] //从 DS 寄存器取出值0x0作为[0x7c64]内存单元的段地址，合起来就是:0:7C64。内存地址0:7C64所指位置代表的是启动盘：从中加载内核的磁盘，0xff表示使用启动盘。指令mov al,[0x7c64]的意思就是将AX低字节AL寄存器的值设为0xff

00000086 3CFF cmp al,0xff //cmp al,0xff指令的意思是，比较AL寄存器的值和0xff的大小,此时AL=0xff，cmp结果相等，即是flag寄存器标记位ZF=1;

00000088 7402 jz 0x8c //jz指令根据flag寄存器标记位ZF是否等于1来进行转移。此时ZF=1，jz指令进行转移，将IP的值设置为0x8c。这里检查我们是否有强制磁盘引用。“强制磁盘引用”通常指的是在引导过程中检查磁盘的特定引导标志或签名，以确定是否找到了有效的引导扇区或引导记录。

0000008C 52 push dx //push dx执行：1.取出寄存器记录的栈顶地址SS:SP=00:2000 2.判断数据宽度：DX寄存器数据是16位，所以SP=SP-2=0X1FFE 3.取出 DX 寄存器的值，由上面00000072 B280 mov dl,0x80指令可知 DX 的低位寄存器 dl 被设置过值0x80， 而整个 DX = 0x0080 4.将 DX 中的内容送入 SS:SP 指向的内存单元处，SS:SP 此时指向新栈顶（0x0080）

该过程只是实际上是调用message过程在屏幕上打印GRUB（或者是Ubuntu？）字样

****

0000008D BB1704 mov bx,0x417 //寄存器赋值：BX=0x417，这一步是将 BX 寄存器设置为磁盘地址包（硬盘的分区表，这个地址很可能是分区表的起始地址。） BX寄存器是4个可以用在[...]中来进行内存单元寻址的寄存器之一。其他三个分别是：SI、DI、BP。只要在[...]中使用寄存器 BP，而指令中没有显性地给出段地址，段地址就默认在 SS 中，除此之外，段地址默认在 DS 中。

00000090 F60703 test byte [bx],0x3 //将 [bx] 内存地址处的值与 0x3 进行按位与操作，并设置标志寄存器的相应标志位。

00000093 7406 jz 0x9b //jz指令根据flag寄存器标记位ZF是否等于1来进行转移。如果此时ZF=1，jz指令进行转移，将IP的值设置为0x9b

00000095 BE887D mov si,0x7d88 //假如ZF不为1，则si0x7d88，并继续执行下面的call

00000098 E81701 call 0x1b2 //call 0x1b2` 指令是调用地址为 `0x1b2` 处的子程序（或函数）。在x86汇编中，`call` 指令用于将当前指令的下一条指令的地址（即 `call` 指令后面紧跟着的地址0000009B  BE057C  mov si,0x7c05 压入栈中，并将控制转移至指定的目标地址。这样做的目的是为了在调用子程序后，能够通过 `ret` 指令将控制返回到 `call` 指令的下一条指令的地址，继续执行后续的指令。在这里，`call 0x1b2` 指令的作用是跳转到地址为 `0x1b2` 的子程序中执行，执行完子程序后会返回到 `call` 指令的下一条指令继续执行。具体子程序中的内容需要根据地址 `0x1b2` 处的代码来确定。这里显然可能是非自动操作或者异常而进行的call，具体0x1b2，在下面引用处继续分析

****

0000009B BE057C mov si,0x7c05 //SI=0x7C05这一步是将 si 寄存器设置为磁盘地址包的地址

0000009E B441 mov ah,0x41 //AH=0x41,ah寄存器代表的是即将执行的中断例程的功能号.int 13 ah=41代表的是，检查是否支持LBA寻址模式。

***

class Two。判断磁盘模式，CHS还是LBA

000000A0 BBAA55 mov bx,0x55aa //BX=0X55AA

000000A3 CD13 int 0x13 //int 13h ah=41H,这个调用检验对特定的驱动器是否存在扩展功能．如果进位标志置 1 则此驱动器不支持扩展功能．如果进位标志为 0，同时 BX = AA55h，则存在扩展功能．

000000A5 5A pop dx //出栈：恢复DX=0x0080

000000A6 52 push dx //入栈：保存DX=0x0080,%dl 可能已被 INT 13 破坏，AH=41H。所以通过重复出入栈

来纠正。

000000A7 723D jc 0xe6 //jc：标志位CF=1则跳转否则不跳转,jc指令与上面的int 13H ah=41H中断例程形成配合。后者的作用是判断 BIOS 是否支持扩展int13中断，如果支持，则CF=0，不跳转。那么jc指令就可以根据 BIOS 是否支持扩展int13中断来执行不同位置的“子程序指令”。

000000A9 81FB55AA cmp bx,0xaa55 //此时 BX=0xAA55，与第二个立即数0xAA55相等,所以标志寄存器ZF更新为1

000000AD 7537 jnz 0xe6 //此时ZF不为0，故不跳转。谨慎起见，这条指令和上一条指令配合使用，继jc 0xe6之后，再次对int 13H ah=41H中断指令的结果进行确认。确认BIOS支持扩展int13。0xe6 处的指令是 CHS 寻址模式的，即是说如果不支持 LBA 寻址模式，则使用 CHS。

000000AF 83E101 and cx,byte +0x1 //byte +0x1 = 0000 0001,由于 int13 ah=41 检查是否支持 LBA了，不知道怎么了CX的最低位被设置了1，就代表支持LBA。并且and指令执行完毕后，cx=0x01,此时，ZF=0

000000B2 7432 jz 0xe6 //此时标志位ZF=0,所以不进行跳转

class Three。使用LBA模式读取start程序，读取到内存0x7000处（stage1加载位于第二扇区的start程序，然后start以磁盘扇区形式而非文件系统形式载入stage2。）

000000B4 31C0 xor ax,ax // AX=0x0,ZF=1

000000B6 894404 mov [si+0x4],ax //上面有指令将 si 寄存器设置为磁盘地址包的地址0x7c05,[si+0x4]作为内存偏移地址，而默认段地址是ds寄存器的值。ds=0x0，所以内存地址就是00:7c09,已知AX=0x00,把AX寄存器的值（16位），写入到内存00:7c09处的连续两个字节。那么这一步就是要把扩展功能的主版本号(在下面引用说明）存入内存。

000000B9 40 inc ax //AX=0x01

000000BA 8844FF mov [si-0x1],al //

000000BD 894402 mov [si+0x2],ax

000000C0 C7041000 mov word [si],0x10 //[si]和ds寄存器，共同表示内存地址=00:7c05,将立即数0x10写入内存

000000C4 668B1E5C7C mov ebx,[0x7c5c] //ebx=0x00000001，实际上就是第二扇区，就是start程序在的地方

/*MBR占据了硬盘的第0个扇区（以LBA方式的逻辑来看，扇区从第0开始编号，若是以物理CHS方式的逻辑来看，扇区便是从第1开始编号）*/

000000C9 66895C08 mov [si+0x8],ebx

000000CD 668B1E607C mov ebx,[0x7c60] //ebx=0x00000000 计算扇区的LBA绝对地址

000000D2 66895C0C mov [si+0xc],ebx

000000D6 C744060070 mov word [si+0x6],0x7000

***

从0B4到0D6完成了内存（磁盘地址包内容）的填写：

00:7c14H 00:7c13H 00:7c12H 00:7c11H 00:7c10H 00:7c0fH 00:7c0eH 00:7c0dH 00:7c0cH

00:7c0bH 00:7c0aH 00:7c09H 00:7c08H 00:7c07H … 00:7c05H 00:7c04H

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x70 0x00 0x00 0x00 0x00

0x01 … 0x10 0x01

***

000000DB B442 mov ah,0x42 //下一条中断功能号ah=42H

/*关于MBR如何寻找到活动分区?

在传统的MBR分区方案中，活动分区通常标记为编号为80h（十六进制）的编号为80h.所以，如果用户不更改活动分区，那么默认情况下，活动分区的编号通常是80h。但请注意，这并不是一项硬性规定，用户可以随时更改活动分区。在多个硬盘的情况下，80h的驱动器可能指向第一个硬盘的第一个分区，但在某些情况下，用户可以通过BIOS设置或操作系统的引导顺序来更改这种映射。所以默认情况下，MBR并没有那种程序性的“寻找”活动分区的过程，而是磁盘地址包向BOIS传的某些固定的信息，活动分区在编号为80h的驱动器*/

000000DD CD13 int 0x13 //读取磁盘：

在磁盘地址包结构中设置适当的值

设置 DS:SI -> 内存中的磁盘地址包(引用)

/*DS:SI = 00:7c05，代表“磁盘地址包”的地址，向BIOS描述磁盘操作的详细信息，根据上面的指令填写得到的磁盘地址包内容分析，传达的信息是：

将编号80（在计算机中，编号为 80 的驱动器通常指的是 BIOS 中的第一个硬盘驱动器。这是由于 BIOS 中的硬盘驱动器编号是从 0x80 开始的）的驱动器的第一个扇区（512 Byte）读到内存地址0x7000（缓存，后面会再移动的）处

```
注意：文档本身所描述的汇编代码是存在内存0x7c00处，地址不同，不要混淆，也不会覆盖。*/
设置 AH = 0x42
设置 DL = “驱动器号”—“C”驱动器通常为 0x80
发出一个 INT 0x13.

000000DF  7205              jc 0xe6  //假设int 13H ah=42H读取磁盘数据到内存成功执行了，那么CF标志位复位0,所以jc 0xe6不进行跳转,0xe6是指当磁盘读取失败，则改用CHS寻址模式再尝试。
000000E1  BB0070            mov bx,0x7000 //BX=0x7000
000000E4  EB76              jmp short 0x15c  // 跳到0x15c，跳转到移动数据到指定位置的调用入口

class Four。将start程序从0x7000移动到指定的启始地址位置，在这里是0x8000，并跳转到start程序

0000015C  60                pusha //pusha: push all,通用寄存器压（按规定顺序）入栈
0000015D  1E                push ds //将寄存器DS压入栈，将此时的ds保存，因为后续有指令对其赋值更新，压入栈以便再次恢复。
0000015E  B90001            mov cx,0x100 //寄存器赋值：CX=0x100，作为后续rep指令的重复次数
00000161  8EDB              mov ds,bx //已知BX=0x7000,寄存器赋值DS=0x7000
00000163  31F6              xor si,si //DS:SI = 7000:00
00000165  BF0080            mov di,0x8000 //寄存器赋值DI=0x8000,0x8000是GRUB引导机内核地址(即操作系统内核（比如Linux内核）在磁盘上的位置，以便GRUB能够加载该内核并将控制权转交给它)
00000168  8EC6              mov es,si //已知：SI=0x00,寄存器赋值ES=0x00
0000016A  FC                cld //已知：SI=0x00,DI=0x8000,标志寄存器更新：DF=0
0000016B  F3A5              rep movsw //将7000:00至7000:100之间的内存数据，完整传送到00:8000至00:8100之间。
0000016D  1F                pop ds //恢复DS寄存器的值，消除本小段代码对寄存器值的破坏
0000016E  61                popa //恢复各通用寄存器的值，消除本小段代码对寄存器值的破坏
0000016F  FF265A7C          jmp [0x7c5a] //根据磁盘位置找到[0x7c5a]的值是0x8000,开始执行在00:8000处的内核指令。[0x7c5a]的值是固定的应该，因此一般会把活动分区的内核复制到00:8000执行?
```