操作系统实验报告--改写 MBR

智能科学与技术　　2024 年 3 月 21 日

实验任务

在分析完整 MBR 的基础上，使用 nasm 实现一个类似 LILO 的交互式引导程序用于替换原 MBR 中的引导代码，其主要功能：1）列出可供选择的启动分区（windows 分区、linux 分区）列表；2）用户可用上下键或数字键进行选择（回车键完成选择）；3）根据用户选择加载指定分区上的操作系统（或引导程序）。

1　汇编语言源代码设计过程

1.1　基本实现思路

根据实验任务，将任务分为三部分的实现：

（1）用户选择界面，利用 BIOS 提供的基本输入输出功能：int10（屏幕输出），int 13（磁盘访问），int 16（键盘输入），输出选择界面，并根据用户选择跳转到对应位置执行汇编码

（2）加载 Windows 的二阶段加载器

（3）加载 Linux 的二阶段加载器

1.2　对 Linux 和 Windows MBR 的分析与提取

1.2.1　Windows

经过分析 Windows 的完整 MBR，可以将 MBR 进行的主要工作分为以下几步：

（1）移动 MBR 从 0x7c00 到 0x600，为活动分区的引导扇区腾出位置，并跳转到新位置继续执行剩余的 MBR 代码

（2）扫描分区表，找到一个激活 (可引导) 分区，并查剩余分区的启动标志是否为 0

（3）测试是否支持 int 0x13 扩展功能，并根据结果分别跳转到传统 int 13h 读取硬盘（CHS）和使用扩展 int 13h 读取硬盘（LBA）

（4）将活动分区的引导扇区装载到内存 7C00 处；

（5）将控制权交给引导扇区代码；

考虑到 MBR 需要载入两种加载器且字节有限，根据上一节实验对已装好硬盘的研究与分析，在已知硬盘与分区表功能与格式正常的前提下，选择省去检查部分，通过分析原硬盘的分区表以及对比硬盘的 Device 情况，可以知道 Windows 的二阶段加载器在分区表第一个分区（同时也是 80 开头的活动分区）的引导扇区，省去寻查部分，采用硬编码的形式实现 Windows 的二阶段加载器的加载。根据以上信息，结合原始 Windows 的 MBR 的分析结果，得到以下比较重要的两部分：

```
1 /*移动MBR并跳转到新位置*/
2 00000000 33C0 xor ax,ax //设置堆栈段地址为0000
3 00000002 8ED0 mov ss,ax
```

```
00000004 BC007C mov sp,0x7c00
00000007 8EC0 mov es,ax
00000009 8ED8 mov ds,ax
0000000B BE007C mov si,0x7c00
0000000E BF0006 mov di,0x600 //DI = 0600
00000011 B90002 mov cx,0x200 //移动 512 个word (512*2 bytes)
00000014 FC cld //清除处理器状态寄存器中的方向标志位（DF）。当方向标志位被清除时，字符串操作（如MOVS、LODS、STOS等）会在每次操作后自动增加索引寄存器（如SI、DI）。
00000015 F3A4 rep movsb
/*
DS:SI= 00:7c00
ES:DI= 00:0600
CX=0x200
DF=0
rep movsb 将MBR从0x7c00移到0x0600
    */
00000017 50 push ax
00000018 681C06 push word 0x61c //将一个16位的值 0x61c 压入栈中
0000001B CB retf //从堆栈中弹出两个字，并将它们作为返回地址加载到指令指针 (IP) 和代码段寄存器 (CS) 中，这里应该是00:061c
0000001C FB sti //执行 "sti" 指令后，处理器将允许中断发生。


/* 使用 int 0x13 扩展功能读 disk */
00000061 666800000000 push dword 0x0
00000067 66FF7608 push dword [bp+0x8]
0000006B 680000 push word 0x0
0000006E 68007C push word 0x7c00 //传输缓冲区地址
00000071 680100 push word 0x1 //读取扇区的个数
00000074 681000 push word 0x10 //数据包尺寸16字节
00000077 B442 mov ah,0x42 //读磁盘选择
00000079 8A5600 mov dl,[bp+0x0] //驱动器号
0000007C 8BF4 mov si,sp //buffer_packet 的地址
0000007E CD13 int 0x13
```

1.2.2 Linux

经过分析 Linux 的完整 MBR，可以将 MBR 进行的主要工作分为以下几步：

（1）启动磁盘的检查以及载入 start 程序前的准备：建栈以及检查是否为硬盘驱动

（2）判断磁盘模式，CHS 还是 LBA

（3）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。）

（4）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

同样考虑到 MBR 需要载入两种加载器且字节有限，根据上一节实验对已装好硬盘的研究与分析，在已知是硬盘驱动以及 BIOS 支持扩展 int 13h（LBA）的前提下，选择省去检查部分，并且通过分析

Linux 的 MBR 反汇编代码，可以知道 Linux 的二阶段加载器存储在硬盘的第二个扇区，采用硬编码的形式实现 Linux 的二阶段加载器的加载。根据以上信息，结合 Linux 原始 MBR 的分析结果，得到以下比较重要的三部分：

```
/*建栈*/
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。


/*使用LBA模式读取start程序，读取到内存0x7000处*/
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
000000DB B442 mov ah,0x42
000000DD CD13 int 0x13

```

```
32
33 /*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*/
34 0000015C 60 pusha //pusha: push all,通用寄存器压（按规定顺序）入栈
35 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。
36 0000015E B90001 mov cx,0x100 //寄存器赋值：CX=0x100，作为后续rep指令的重复次数
37 00000161 8EDB mov ds,bx //已知BX=0x7000,寄存器赋值DS=0x7000
38 00000163 31F6 xor si,si //DS:SI = 7000:00
39 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)
40 00000168 8EC6 mov es,si //已知：SI=0x00,寄存器赋值ES=0x00
41 0000016A FC cld //已知：SI=0x00,DI=0x8000,标志寄存器更新：DF=0
42 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。
43 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
44 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
45 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。
```

2 编写 MBR

2.1 实验环境与准备

1. 上一节实验使用 QEMU 创建的装载了 Windows7 与 Ubuntu 双系统的虚拟硬盘。（虚拟硬盘总 55G，其中 Windows7 占 30G，Ubuntu 占 25G）

2. Vscode 的 Hex Editor 工具，用于修改硬盘的磁盘签名与分区表

3. GDB 调试工具

2.2 汇编语言源代码以及 MBR

根据基本实现思路，分三步分别实现用户选择界面，加载 Windows 以及加载 Linux 的汇编语言源代码并进行测试与验证，最终将三部分组合成完整的 MBR 引导程序。

2.2.1 编写过程遇到的实验困难与解决

1. 在编写 Windows 的拉取代码时，在一开始移动了 MBR 后没有跳转到新位置执行，导致 MBR 继续在原位置往下执行。将 Windows 二阶段加载器读取到 0x7c00 后覆盖了原 MBR 且 IP 仍在原地址继续往下执行，导致错误。

2. 移动了 MBR 后跳转到新位置执行，但是新位置编写错误，需要反汇编 new.bin 查看下一条代码的偏移位置。

3. 将数据段放在了汇编源代码的最前面，导致无法正确拉起 Windows，BIOS 默认从一开始执行的就是指令，导致错误。将数据段放在最后，实验成功。

2.2.2 汇编语言源代码

mbr.asm

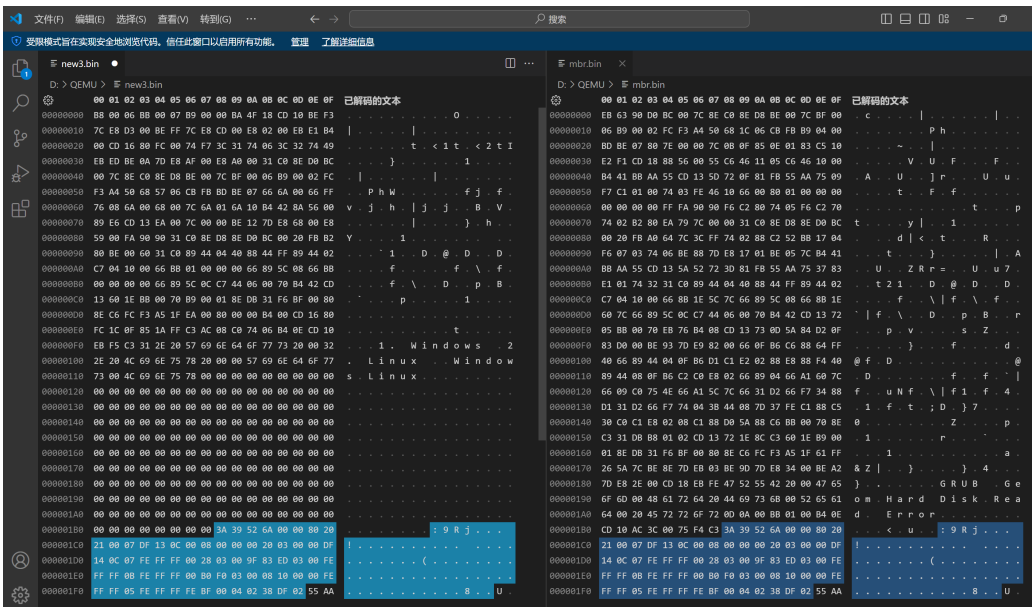
```
; 设置代码段的起始位置
SECTION MBR vstart=0x7c00
_start:
    ; 清屏
    mov ax, 0600h
    mov bx, 0700h
    mov cx, 0
    mov dx, 184fh
    int 10h

    ; 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

    ;按数字‘1’键，选择Windows，再等待回车或者改变选择
    cmp al, '1'
    je windows
    ;按数字‘2’键，选择Linux，再等待回车或者改变选择
    cmp al, '2'
    je linux
    ;非法选择，重新选择
    jmp handle_input


; 用户选择 Windows 分区的操作
windows:
    mov si, message_windows
    call print_string
```

```
44      call wait_for_enter ; 等待用户按下回车键
45
46      ;把MBR从0x7c00移到0x600
47      xor ax, ax
48      mov ss, ax
49      mov sp, 0x7c00
50      mov es, ax
51      mov ds, ax
52      mov si, 0x7c00
53      mov di, 0x600
54      mov cx, 0x200
55      cld
56      rep movsb
57
58  ;注意，我们要跳转到新地址继续执行MBR剩余的代码，0x662是sti这条命令的新地址，注意是0x600
        加上偏移（即第几个字节）
59      push ax
60      push word 0x657
61      retf
62
63  ;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第一个扇区，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），下面是扩展的
        int 13h（LBA）的方法，也可以用CHS
64      sti
65      mov bp, 0x7be
66      push dword 0x0
67      push dword [bp+0x8]
68      push word 0x0
69      push word 0x7c00
70      push word 0x1
71      push word 0x10
72      mov ah,0x42
73      mov dl,[bp+0x0]
74      mov si,sp
75      int 0x13
76      jmp 00:0x7c00
77
78  ; 用户选择 Linux 分区的操作
79  linux:
80      mov si, message_linux
81      call print_string
82      call wait_for_enter ; 等待用户按下回车键
83
84  ; 建栈
85      cli
86      nop
87      nop
```

```
88      xor ax, ax
89      mov ds, ax
90      mov ss, ax
91      mov sp, 0x2000
92      sti
93      mov dl, 0x80
94
95  ;使用LBA模式读取start程序，读取到内存0x7000处（从Linux原MBR提取代码）
96      mov si,0x6000
97      xor ax,ax
98      mov [si+0x4],ax
99      inc ax
100     mov [si-0x1],al
101     mov [si+0x2],ax
102     mov word [si],0x10
103     mov ebx,0x00000001
104     mov [si+0x8],ebx
105     mov ebx,0x00000000
106     mov [si+0xc],ebx
107     mov word [si+0x6],0x7000
108     mov ah,0x42
109     int 0x13
110
111 ;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（从
        Linux原MBR提取代码）
112     pusha
113     push ds
114     mov bx,0x7000
115     mov cx,0x100
116     mov ds,bx
117     xor si,si
118     mov di,0x8000
119     mov es,si
120     cld
121     rep movsw
122     pop ds
123     jmp 00:0x8000
124
125 ; 等待用户按下回车键
126 wait_for_enter:
127     mov ah, 0 ; 重置AH为0
128     int 16h ; 调用BIOS中断等待键盘输入
129     cmp ah, 0x1C ; 检查是否为回车键
130     jne _start ; 按下任意键可以重新选择
131     ret ; 如果是回车键，则返回
132
133 ; 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，需要先保存原本硬盘的 bin 文件 (mbr.bin)，然后使用 nasm 编译 mbr.asm 为 new.bin 后，再到 vscode 利用 hex editor，把原硬盘 mbr.bin 文件的从 441 开始到最后的字节拷贝过来修改 new.bin。



相关命令行：

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

2.2.3 nasm 格式反汇编以及完整 MBR

```
00000000 B80006 mov ax,0x600
00000003 BB0007 mov bx,0x700
00000006 B90000 mov cx,0x0
00000009 BA4F18 mov dx,0x184f
0000000C CD10 int 0x10
0000000E BEF37C mov si,0x7cf3
00000011 E8D300 call 0xe7
00000014 BEFF7C mov si,0x7cff
00000017 E8CD00 call 0xe7
0000001A E80200 call 0x1f
0000001D EBE1 jmp short 0x0
0000001F B400 mov ah,0x0
00000021 CD16 int 0x16
00000023 80FC00 cmp ah,0x0
00000026 74F7 jz 0x1f
00000028 3C31 cmp al,0x31
0000002A 7406 jz 0x32
0000002C 3C32 cmp al,0x32
0000002E 7449 jz 0x79
00000030 EBED jmp short 0x1f
00000032 BE0A7D mov si,0x7d0a
00000035 E8AF00 call 0xe7
00000038 E8A000 call 0xdb
0000003B 31C0 xor ax,ax
0000003D 8ED0 mov ss,ax
0000003F BC007C mov sp,0x7c00
00000042 8EC0 mov es,ax
00000044 8ED8 mov ds,ax
00000046 BE007C mov si,0x7c00
00000049 BF0006 mov di,0x600
0000004C B90002 mov cx,0x200
0000004F FC cld
00000050 F3A4 rep movsb
00000052 50 push ax
00000053 685706 push word 0x657
00000056 CB retf
00000057 FB sti
```

```
00000058 BDBE07 mov bp,0x7be
0000005B 666A00 o32 push byte +0x0
0000005E 66FF7608 push dword [bp+0x8]
00000062 6A00 push byte +0x0
00000064 68007C push word 0x7c00
00000067 6A01 push byte +0x1
00000069 6A10 push byte +0x10
0000006B B442 mov ah,0x42
0000006D 8A5600 mov dl,[bp+0x0]
00000070 89E6 mov si,sp
00000072 CD13 int 0x13
00000074 EA007C0000 jmp 0x0:0x7c00
00000079 BE127D mov si,0x7d12
0000007C E86800 call 0xe7
0000007F E85900 call 0xdb
00000082 FA cli
00000083 90 nop
00000084 90 nop
00000085 31C0 xor ax,ax
00000087 8ED8 mov ds,ax
00000089 8ED0 mov ss,ax
0000008B BC0020 mov sp,0x2000
0000008E FB sti
0000008F B280 mov dl,0x80
00000091 BE0060 mov si,0x6000
00000094 31C0 xor ax,ax
00000096 894404 mov [si+0x4],ax
00000099 40 inc ax
0000009A 8844FF mov [si-0x1],al
0000009D 894402 mov [si+0x2],ax
000000A0 C7041000 mov word [si],0x10
000000A4 66BB01000000 mov ebx,0x1
000000AA 66895C08 mov [si+0x8],ebx
000000AE 66BB00000000 mov ebx,0x0
000000B4 66895C0C mov [si+0xc],ebx
000000B8 C744060070 mov word [si+0x6],0x7000
000000BD B442 mov ah,0x42
000000BF CD13 int 0x13
000000C1 60 pusha
000000C2 1E push ds
000000C3 BB0070 mov bx,0x7000
000000C6 B90001 mov cx,0x100
000000C9 8EDB mov ds,bx
000000CB 31F6 xor si,si
000000CD BF0080 mov di,0x8000
000000D0 8EC6 mov es,si
000000D2 FC cld
```

```
000000D3 F3A5 rep movsw
000000D5 1F pop ds
000000D6 EA00800000 jmp 0x0:0x8000
000000DB B400 mov ah,0x0
000000DD CD16 int 0x16
000000DF 80FC1C cmp ah,0x1c
000000E2 0F851AFF jnz near 0x0
000000E6 C3 ret
000000E7 AC lodsb
000000E8 08C0 or al,al
000000EA 7406 jz 0xf2
000000EC B40E mov ah,0xe
000000EE CD10 int 0x10
000000F0 EBF5 jmp short 0xe7
000000F2 C3 ret
000000F3 312E2057 xor [0x5720],bp
000000F7 696E646F77 imul bp,[bp+0x64],word 0x776f
000000FC 7320 jnc 0x11e
000000FE 0032 add [bp+si],dh
00000100 2E204C69 and [cs:si+0x69],cl
00000104 6E outsb
00000105 7578 jnz 0x17f
00000107 2000 and [bx+si],al
00000109 005769 add [bx+0x69],dl
0000010C 6E outsb
0000010D 646F fs outsw
0000010F 7773 ja 0x184
00000111 004C69 add [si+0x69],cl
00000114 6E outsb
00000115 7578 jnz 0x18f
...
000001FE 55 push bp
000001FF AA stosb
```

完整 MBR 已在作业附件 MBR.docx 中提交

2.3 测试结果

用户交互界面正常，用户按下 1 键，屏幕出现 Windows，按除了回车键的任意键可以重新选择，按下回车则拉起 Windows7；用户按下 2 键，屏幕出现 Linux，按除回车键的任意键可以重新选择，按下回车则拉起 Ubuntu；

Figure 1: 用户选择界面

Figure 2: 用户按下 1 键

Figure 3: 用户按下 2 键

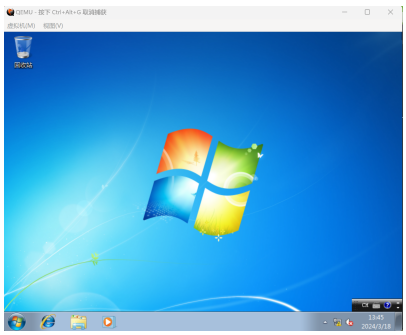


Figure 4: 拉起 Windows7 成功

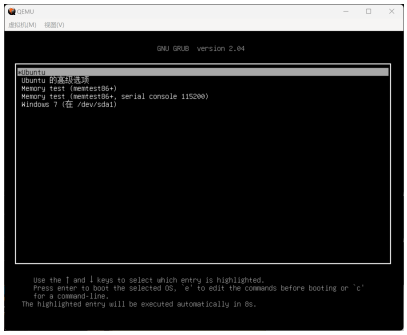


Figure 5: 拉起 Ubuntu 成功

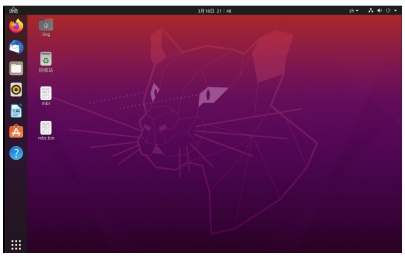


Figure 6: 进入 Ubuntu

3 参考文章

1. 反汇编 mbr(windows)
2. MBR(主引导记录) 的反汇编 + 注释 (windows)
3. bootloader 详解
4. 键盘 I/O 中断调用（INT 16H）和常见的 int 17H、int 1A H
5. 操作系统实现——编写 MBR
6. Windows 启动过程 (MBR 引导过程分析)