

# 实验2 第1部分

- 实验2 第1部分
  - 个人信息
  - 实验名称
  - 实验目的
  - 实验要求
  - 实验环境
    - 1. 虚拟机
    - 2. WSL
  - 实验过程
    - 练习3 分析 bootloader 进入保护模式的过程
      - 1. 宏定义
      - 2. 寄存器清零
      - 3. 开启 A20
        - 阶段1
        - 阶段2
      - 4. 开启保护模式
      - 5. 执行32位代码
      - 6. 初始化 gdt
    - 练习4 分析 bootloader 加载 ELF 格式的 OS 过程
      - 1. bootloader 如何读取硬盘扇区的
      - 2. bootloader 如何加载 ELF 格式的 OS 的
    - 练习5 实现函数调用堆栈跟踪函数
    - 练习6 完善中断初始化和处理
      - 中断向量表中一个表项占多少字节
      - 其中哪几位代表中断处理代码的入口
      - 完善 trap.c 中的 idt\_init()
      - 完善 trap.c 中的 trap()
  - 实验结果
  - 实验总结
    - 问题1: 尝试在 WSL 环境下运行相应代码

## 个人信息

---

- 数据科学与计算机学院
- 2018级 米家龙

## 实验名称

---

## 实验目的

---

- 熟悉并掌握 Ubuntu 系统的相关操作
- 了解并掌握硬件模拟器 Qemu 的操作
- 熟悉并掌握 gdb 的相关操作
- 学会使用 Qemu 和 gdb 运行和调试程序
- 了解 bootloader 进入保护模式的过程
- 了解 bootloader 加载 OS 的过程
- 了解并熟悉操作系统的中断和堆栈调用

## 实验要求

---

1. 练习3：分析 bootloader 从实模式进入保护模式的过程
2. 练习4：分析 bootloader 加载 ELF 格式 OS 的过程
  1. bootloader 如何读取硬盘扇区的？
  2. bootloader 如何加载 ELF 格式的 OS 的？
3. 练习5：实现函数调用堆栈跟踪函数
4. 练习6：完善中断初始化和处理

## 实验环境

---

### 1. 虚拟机

使用老师提供的 `mooc-os-2015.vdi`，在虚拟机中创建 64 位的 Ubuntu 虚拟机并加载该 vdi，获得了版本为：

```
Linux moocos-VirtualBox 3.13.0-24-generic #46-Ubuntu SMP Thu Apr 10 19:11:08 UTC 2014
x86_64 x86_64 x86_64 GNU/Linux
```

的虚拟机操作系统

### 2. WSL

WSL 配置如下：

```
root@LAPTOP-QTCGESH0:/mnt/d/blog/work/matrix/step1/001# uname -a
Linux LAPTOP-QTCGESH0 4.4.0-19041-Microsoft #1-Microsoft Fri Dec 06 14:06:00 PST 2019
x86_64 x86_64 x86_64 GNU/Linux
```

## 实验过程

---

### 练习3 分析 bootloader 进入保护模式的过程

从前两个练习中可以分析得到，bootloader 在接手 BIOS 之后，开始执行 `bootasm.S` 中的源码，此时系统处于实模式的运行状态，地址为16位，需要通过修改 A20 地址线进行引导进入保护模式，从而扩大地址到32位并且内存扩大到4G

查看 `bootasm.S` 的代码，了解 bootloader 是如何从实模式转到保护模式的

## 1. 宏定义

```
.set PROT_MODE_CSEG,      0x8           # kernel code segment selector
.set PROT_MODE_DSEG,      0x10        # kernel data segment selector
.set CR0_PE_ON,           0x1         # protected mode enable flag
```

上述代码通过宏定义了内核代码段选择子、内核数据段选择子以及保护模式标志位

## 2. 寄存器清零

```
# start address should be 0:7c00, in real mode, the beginning address of the running
bootloader
.globl start
start:
.code16                                # Assemble for 16-bit mode
    cli                                # Disable interrupts
    cld                                # String operations increment

    # Set up the important data segment registers (DS, ES, SS).
    xorw %ax, %ax                      # Segment number zero
    movw %ax, %ds                      # -> Data Segment
    movw %ax, %es                      # -> Extra Segment
    movw %ax, %ss                      # -> Stack Segment
```

上述代码中，设置全局标识符为 `start`，在该代码中，先设置了运行模式为16位，关闭中断并清除了方向标志

之后将代码段寄存器（DS）、附加段寄存器（ES）和堆栈段寄存器（SS）清零

## 3. 开启 A20

如前面所受，为了进入保护模式，需要修改 A20 地址线为1，从而开启 A20，修改 A20 是因为：

未开启 A20 时，可访问的物理内存空间为 1MB，任何地址高于 1MB 的都将被置为 `0x000000`，32位 CPU 所能够管理的 4G 内存无法使用

启用 A20 分为两个阶段

### 阶段1

```
# Enable A20:
# For backwards compatibility with the earliest PCs, physical
# address line 20 is tied low, so that addresses higher than
# 1MB wrap around to zero by default. This code undoes this.
seta20.1:
    # Wait for not busy(8042 input buffer empty).
    inb $0x64, %al                    # 获取状态寄存器值，存入 al 中
    testb $0x2, %al                  # 比较首位是否为1
    jnz seta20.1                      # 如果为0，则跳转到 seta20.1

    movb $0xd1, %al                   # 0xd1 -> port 0x64
    outb %al, $0x64                  # 0xd1 means: write data to 8042's
P2 port
```

该阶段有两个步骤：

- 等待8042缓冲区为输入为空
- 通过写命令将数据写到8042的 P2 端口

## 阶段2

```
seta20.2:
    # Wait for not busy(8042 input buffer empty).
    inb $0x64, %al                # 获取状态寄存器值，存入 al 中
    testb $0x2, %al              # 比较首位是否为1
    jnz seta20.2                 # 如果为0，跳转到 seta20.2

    movb $0xdf, %al              # 0xdf -> port 0x60
    outb %al, $0x60              # 0xdf = 11011111, means set P2's
                                # A20 bit(the 1 bit) to 1
```

- 等待8042缓冲区输入为孔
- 通过写命令将数据传送到8042缓冲区，从而开启 A20

## 4. 开启保护模式

```
# Switch from real to protected mode, using a bootstrap GDT
# and segment translation that makes virtual addresses
# identical to physical addresses, so that the
# effective memory map does not change during the switch.
lgdt gdtdesc                    # 加载 gdt
movl %cr0, %eax                 # 加载 cr0 到 eax
orl $CR0_PE_ON, %eax           # 将 eax 的第0位置0
movl %eax, %cr0                # 使 cr0 的第0位为0

# Jump to next instruction, but in 32-bit code segment.
# Switches processor into 32-bit mode.
ljmp $PROT_MODE_CSEG, $protcseg
```

该过程分为4个步骤：

- 加载 gdt
- 段变换，和 gdt 一起使得虚拟地址和物理地址相同，避免在切换模式的过程中改变有效内存映射
- 将 cr0 中的保护允许位（PE, Protect Enable）置1，从而开启保护模式
- 跳转到32位代码段，使得处理器切换到32位模式

## 5. 执行32位代码

```
.code32                        # Assemble for 32-bit mode
protcseg:
    # Set up the protected-mode data segment registers
    movw $PROT_MODE_DSEG, %ax  # Our data segment selector
    movw %ax, %ds              # -> DS: Data Segment
    movw %ax, %es              # -> ES: Extra Segment
    movw %ax, %fs              # -> FS
    movw %ax, %gs              # -> GS
    movw %ax, %ss              # -> SS: Stack Segment

    # Set up the stack pointer and call into C.
    # The stack region is from 0--start(0x7c00)
    movl $0x0, %ebp
    movl $start, %esp          # start 是 0x7c00
```

```

    call bootmain

    # If bootmain returns (it shouldn't), loop.
spin:
    jmp spin

```

在32位模式下，`protcseg` 汇编代码段主要执行了：

- 将抱歉16位模式下的3个寄存器在内的合计5个寄存器置0
- 设置栈指针并调用 C 代码，转到 `bootmain`，进入保护模式完成

如果 `bootmain` 返回了，则进行循环

## 6. 初始化 gdt

```

# Bootstrap GDT
.p2align 2                                # 后移位置计数器，置4字节，用于内存对齐
gdt:
    SEG_NULLASM                            # null seg
    # code seg for bootloader and kernel
    # 将 0x0 ~ 0xffffffff 设置为 可执行段 或 可读可执行段
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)
    # data seg for bootloader and kernel
    # 将 0x0 ~ 0xffffffff 设置为可读不可执行段
    SEG_ASM(STA_W, 0x0, 0xffffffff)

gdtdesc:
    .word 0x17                            # sizeof(gdt) - 1，代表 gdt 大小
    .long gdt                             # address gdt

```

## 练习4 分析 bootloader 加载 ELF 格式的 OS 过程

通过阅读 `bootmain.c`，了解 bootloader 如何加载 ELF 文件。通过分析源代码和通过 `qemu` 来运行并调试 `bootloader&OS`

### 1. bootloader 如何读取硬盘扇区的

从查看 `bootmain.S` 的代码中可以得到，当 bootloader 将 CPU 切换为保护模式之后，会调用 `bootmain` 函数，从磁盘上加载并运行 OS，bootloader 的访问硬盘都是 `LBA 模式的 PIO (Program IO)` 方式，即所有的 IO 操作是通过 CPU 访问硬盘的 IO 地址寄存器完成

bootloader 通过 IO 地址寄存器 `0x1f0 - 0x1f7` 读取状态和发出命令。

IO 地址	功能
0x1f0	当 0x1f7 不为忙状态时可读
0x1f2	储存要读写的扇区数量，最少为1
0x1f3	LBA 模式下 LBA 参数的 0-7 位
0x1f4	LBA 模式下 LBA 参数的 8-15 位
0x1f5	LBA 模式下 LBA 参数的 16-23 位
0x1f6	0-3位：LAB 模式下 LBA 参数的 24-27 位； 第4位：0为主盘，1位从盘
0x1f7	状态和命令寄存器，操作时先给命令再读取，如果不是忙状态则从 0x1f0 读取数据

加载相关数据类型和操作的库

```
#include <defs.h>    // 定义类型、指针位数以及对地址的处理
#include <x86.h>      // 定义 I/O 输入输出，处理描述符表的汇编指令，以及字符串处理函数
#include <elf.h>      // 定义 elf 文件头和进程头信息
```

```
// 定义扇区大小
#define SECTSIZE      512
// 定义 ELF 头空间
#define ELFHDR        ((struct elfhdr *)0x10000)
```

```
/* waitdisk - wait for disk ready */
static void
waitdisk(void) {
    while ((inb(0x1F7) & 0xC0) != 0x40)
        /* do nothing */;
}
```

该函数用于同步等待直到硬盘可读

```
/* readsect - read a single sector at @secno into @dst */
static void
readsect(void *dst, uint32_t secno) {
    // wait for disk to be ready
    waitdisk();

    // outb() 是以 LBA 模式的 PIO 方式读写硬盘
    // 一次只能读写一个扇区
    outb(0x1F2, 1); // count = 1 读入的硬盘扇区数量为1
    outb(0x1F3, secno & 0xFF); // 设置 LBA 0 - 7 位
    outb(0x1F4, (secno >> 8) & 0xFF); // 设置 LBA 8 - 15 位
    outb(0x1F5, (secno >> 16) & 0xFF); // 设置 LBA 16 - 23 位
    outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0); // 设置 LBA 24 - 27 位，并设置位主盘
    outb(0x1F7, 0x20); // 命令 0x20，读取数据

    // wait for disk to be ready
    waitdisk();

    // read a sector
    insl(0x1F0, dst, SECTSIZE / 4); // 获取数据
}
```

综上，读取硬盘的流程如下

- 首先通过 0x1f7 状态和指令寄存器判断硬盘是否可读
- 若硬盘可读则向 0x1f7 寄存器中写读指令
- 再次等待可读
- 可读后从 0x1f0 处读取硬盘

## 2. bootloader 如何加载 ELF 格式的 OS 的

```
static void
readseg(uintptr_t va, uint32_t count, uint32_t offset) {
    uintptr_t end_va = va + count;                                // 结束地址

    va -= offset % SECTSIZE;                                       // 向下舍入，得到扇区边

    // 从字节转换到扇区
    // 使得内核将扇区1开始
    uint32_t secno = (offset / SECTSIZE) + 1;

    // 可以实现同时读取多个扇区，前提是该函数太慢
    for (; va < end_va; va += SECTSIZE, secno++) {
        readsect((void *)va, secno);
    }
}
```

该函数用于读取寄存器，从内核的 `offset` 处获取 `count` 个字节的数据到虚拟地址 `va` 中；并且复制得到的内容可能超过 `count` 个字节

`bootmain` 作为主要的执行函数，里面包含了加载 ELF 格式的 OS 的具体过程，但在分析该函数之前，需要查看包含对应数据结构的 `elf.h` 头文件，来了解相关操作的详情

在 `elf.h` 中，定义了两个结构体，分别为 `elfhdr`，代表 elf 文件头，和 `proghdr`，代表程序块头

```
/* file header */
struct elfhdr {
    uint32_t e_magic;        // must equal ELF_MAGIC (0x464C457FU)
    uint8_t e_elf[12];
    uint16_t e_type;         // 1=relocatable, 2=executable, 3=shared object, 4=core
    image
    uint16_t e_machine;      // 3=x86, 4=68K, etc. 架构
    uint32_t e_version;      // file version, always 1
    uint32_t e_entry;        // 程序入口的虚拟地址（如果存在）
    uint32_t e_phoff;        // program header 表位置的偏移
    uint32_t e_shoff;        // section header 表位置的偏移
    uint32_t e_flags;        // architecture-specific flags, usually 0
    uint16_t e_ehsize;       // elf 头大小
    uint16_t e_phentsize;    // 程序头入口的大小
    uint16_t e_phnum;        // 程序头入口的数量
    uint16_t e_shentsize;    // 节入口的大小
    uint16_t e_shnum;        // 节入口的数量
    uint16_t e_shstrndx;     // 包含节名称字符串的节的数量
};

/* program section header */
struct proghdr {
    uint32_t p_type;         // 可加载代码/数据/动态链接信息/其他
    uint32_t p_offset;       // 段偏移量
}
```

```

uint32_t p_va;      // map segment 的虚拟地址
uint32_t p_pa;      // 物理地址，忌用
uint32_t p_filesz;  // 文件中的段大小
uint32_t p_memsz;   // 内存中的段大小，如果包含了 bss，那么会更大
uint32_t p_flags;   // 读/写/执行位
uint32_t p_align;   // 用于对齐代码
};

```

了解了上述两种头类型，再来看 bootmain() 函数

```

/* bootmain - the entry of bootloader */
void
bootmain(void) {
    // 读取磁盘的第一页
    readseg((uintptr_t)ELFHDR, SECTSIZE * 8, 0);

    // 判断是否是有效地 elf 文件
    if (ELFHDR->e_magic != ELF_MAGIC) {
        goto bad;
    }

    struct proghdr *ph, *eph;

    // 加载程序段
    ph = (struct proghdr *)((uintptr_t)ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
    // 按照表的顺序将 elf 文件数据读入内存
    for (; ph < eph; ph++) {
        readseg(ph->p_va & 0xFFFFF, ph->p_memsz, ph->p_offset);
    }

    // 调用 elf 头中的程序入口，并且没有返回值
    ((void (*)(void))(ELFHDR->e_entry & 0xFFFFF))();

    // 用于 elf 无效时的处理
bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);

    /* do nothing */
    while (1);
}

```

综上，bootloader 加载 elf 格式的 os 的过程为：

- 读取磁盘，获得 elf 头
- 通过 elf 头中的 e\_magic 字段来判断是否为合法 elf 文件
- 从 elf 头中得到程序头的地址，从而拿到每个段的相关信息
- 通过循环读取每个段的信息
- 根据段的偏移量将程序段数据读入内存

## 练习5 实现函数调用堆栈跟踪函数

根据提示，代码如下

```

void
print_stackframe(void) {
    uint32_t ebp = read_ebp();           // 获取 ebp
}

```



```

uint32_t eip = read_eip();                                // 获取 eip

int i = 0;
for (i = 0; i < STACKFRAME_DEPTH; i++)
{
    cprintf("ebp = 0x%08x\teip = 0x%08x\n", ebp, eip);
    uint32_t *arguments = (uint32_t *) ebp + 2;           // 拿到参数
    cprintf("\targ0 = 0x%08x\targ1 = 0x%08x\n\targ2 = 0x%08x\targ3 = 0x%08x\n",
            *(arguments), *(arguments + 1), *(arguments + 2), *(arguments + 3));
    print_debuginfo(eip - 1);
    eip = ((uint32_t *)ebp)[1];                             // 将 eip 指向上一个函数栈段
的返回地址
    ebp = ((uint32_t *)ebp)[0];                             // ebp 指向上一个函数栈段的
ebp
}
}

```

通过 `make` 命令能够编译成功

```

+ cc kern/init/stub.c
+ cc kern/debug/kdebug.c
+ cc kern/debug/kmonitor.c
+ cc kern/debug/panic.c
kern/debug/panic.c: In function '__panic':
kern/debug/panic.c:27:5: warning: implicit declaration of function 'print_stackframe' [-Wimplicit-function-declaration]
    print_stackframe();
    ^
+ cc kern/driver/clock.c
+ cc kern/driver/console.c
+ cc kern/driver/intr.c
+ cc kern/driver/picirq.c
+ cc kern/trap/trap.c
kern/trap/trap.c:14:13: warning: 'print_ticks' defined but not used [-Wunused-function]
    static void print_ticks() {
                ^
kern/trap/trap.c:30:26: warning: 'idt_pd' defined but not used [-Wunused-variable]
    static struct pseudodesc idt_pd = {
                            ^
+ cc kern/trap/trapentry.S
+ cc kern/trap/vectors.S
+ cc kern/mm/pmm.c
+ cc libs/printfmt.c
+ cc libs/string.c
+ ld bin/kernel
+ cc boot/bootasm.S
+ cc boot/bootmain.c
+ cc tools/sign.c
+ ld bin/bootblock
'obj/bootblock.out' size: 472 bytes
build 512 bytes boot sector: 'bin/bootblock' success!
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB) copied, 0.0311802 s, 164 MB/s
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.000221822 s, 2.3 MB/s
146+1 records in
146+1 records out
74871 bytes (75 kB) copied, 0.00063311 s, 118 MB/s
mijialong$>

```

因为使用终端，因此使用 `make qemu-nox V=` 来查看输出结果，发现输出次数较多，而且存在 `ebp` 为0的时候仍然在循环，不符合预期

```
mijialong$>make qemu-nox V=
qemu-system-i386 -no-reboot -serial mon:stdio -hda bin/ucore.img -nographic
(THU.CST) os is loading ...
```

Special kernel symbols:

```
entry 0x00100000 (phys)
etext 0x001032d5 (phys)
edata 0x0010ea16 (phys)
end 0x0010fd20 (phys)
```

Kernel executable memory footprint: 64KB

```
ebp = 0x00007b08 eip = 0x001009a7
    arg0 = 0x000010094 arg1 = 0x000000000
    arg2 = 0x00007b38 arg3 = 0x00100092
    kern/debug/kdebug.c:307: print_stackframe+22
ebp = 0x00007b18 eip = 0x00100c94
    arg0 = 0x000000000 arg1 = 0x000000000
    arg2 = 0x000000000 arg3 = 0x00007b88
    kern/debug/kmonitor.c:125: mon_backtrace+10
ebp = 0x00007b38 eip = 0x00100092
    arg0 = 0x000000000 arg1 = 0x00007b60
    arg2 = 0xffff0000 arg3 = 0x00007b64
    kern/init/init.c:48: grade_backtrace2+33
ebp = 0x00007b58 eip = 0x001000bb
    arg0 = 0x000000000 arg1 = 0xffff0000
    arg2 = 0x00007b84 arg3 = 0x00000029
    kern/init/init.c:53: grade_backtrace1+38
ebp = 0x00007b78 eip = 0x001000d9
    arg0 = 0x000000000 arg1 = 0x00100000
    arg2 = 0xffff0000 arg3 = 0x0000001d
    kern/init/init.c:58: grade_backtrace0+23
ebp = 0x00007b98 eip = 0x001000fe
    arg0 = 0x001032fc arg1 = 0x001032e0
    arg2 = 0x0000130a arg3 = 0x00000000
    kern/init/init.c:63: grade_backtrace+34
ebp = 0x00007bc8 eip = 0x00100055
    arg0 = 0x000000000 arg1 = 0x000000000
    arg2 = 0x000000000 arg3 = 0x000010094
    kern/init/init.c:28: kern_init+84
ebp = 0x00007bf8 eip = 0x00007d68
    arg0 = 0xc031fcfa arg1 = 0xc08ed88e
    arg2 = 0x64e4d08e arg3 = 0xfa7502a8
    <unknown>: -- 0x00007d67 --
ebp = 0x00000000 eip = 0x00007c4f
    arg0 = 0xf000e2c3 arg1 = 0xf000ff53
    arg2 = 0xf000ff53 arg3 = 0xf000ff53
    <unknown>: -- 0x00007c4e --
ebp = 0xf000ff53 eip = 0xf000ff53
    arg0 = 0x000000000 arg1 = 0x000000000
    arg2 = 0x000000000 arg3 = 0x000000000
    <unknown>: -- 0xf000ff52 --
ebp = 0x00000000 eip = 0x00000000
    arg0 = 0xf000e2c3 arg1 = 0xf000ff53
    arg2 = 0xf000ff53 arg3 = 0xf000ff53
    <unknown>: -- 0xffffffff --
ebp = 0xf000ff53 eip = 0xf000ff53
    arg0 = 0x000000000 arg1 = 0x000000000
    arg2 = 0x000000000 arg3 = 0x000000000
    <unknown>: -- 0xf000ff52 --
ebp = 0x00000000 eip = 0x00000000
    arg0 = 0xf000e2c3 arg1 = 0xf000ff53
    arg2 = 0xf000ff53 arg3 = 0xf000ff53
    <unknown>: -- 0xffffffff --
ebp = 0xf000ff53 eip = 0xf000ff53
    arg0 = 0x000000000 arg1 = 0x000000000
    arg2 = 0x000000000 arg3 = 0x000000000
    <unknown>: -- 0xf000ff52 --
ebp = 0x00000000 eip = 0x00000000
```

```

    arg0 = 0xf000e2c3    arg1 = 0xf000ff53
    arg2 = 0xf000ff53    arg3 = 0xf000ff53
    <unknown>: -- 0xffffffff --
ebp = 0xf000ff53    eip = 0xf000ff53
    arg0 = 0x00000000    arg1 = 0x00000000
    arg2 = 0x00000000    arg3 = 0x00000000
    <unknown>: -- 0xf000ff52 --
ebp = 0x00000000    eip = 0x00000000
    arg0 = 0xf000e2c3    arg1 = 0xf000ff53
    arg2 = 0xf000ff53    arg3 = 0xf000ff53
    <unknown>: -- 0xffffffff --
ebp = 0xf000ff53    eip = 0xf000ff53
    arg0 = 0x00000000    arg1 = 0x00000000
    arg2 = 0x00000000    arg3 = 0x00000000
    <unknown>: -- 0xf000ff52 --
ebp = 0x00000000    eip = 0x00000000
    arg0 = 0xf000e2c3    arg1 = 0xf000ff53
    arg2 = 0xf000ff53    arg3 = 0xf000ff53
    <unknown>: -- 0xffffffff --
ebp = 0xf000ff53    eip = 0xf000ff53
    arg0 = 0x00000000    arg1 = 0x00000000
    arg2 = 0x00000000    arg3 = 0x00000000
    <unknown>: -- 0xf000ff52 --
++ setup timer interrupts
mijialong$>

```

修改代码为

```

void
print_stackframe(void) {
    uint32_t ebp = read_ebp();           // 获取 ebp
    uint32_t eip = read_eip();           // 获取 eip

    int i = 0;
    for (i = 0; ebp != 0 && i < STACKFRAME_DEPTH; i++) // 新增判断条件 ebp != 0, 用于避免无意义返回
    {
        cprintf("ebp = 0x%08x\teip = 0x%08x\n", ebp, eip);
        uint32_t *arguments = (uint32_t *) ebp + 2; // 拿到参数
        cprintf("\targ0 = 0x%08x\targ1 = 0x%08x\n\targ2 = 0x%08x\targ3 = 0x%08x\n",
                *(arguments), *(arguments + 1), *(arguments + 2), *(arguments + 3));
        print_debuginfo(eip - 1);
        eip = ((uint32_t *)ebp)[1]; // 将 eip 指向上一个函数栈段的返回地址
        ebp = ((uint32_t *)ebp)[0]; // ebp 指向上一个函数栈段的ebp
    }
}

```

通过之前的命令编译和运行后，得到下图结果，可以符合预期

```
74072 bytes (73 KB) copied, 0.000702301 s, 30411 KB/s
mijialong$>make qemu-nox V=
qemu-system-i386 -no-reboot -serial mon:stdio -hda bin/ucore.img -nographic
(THU.CST) os is loading ...
```

Special kernel symbols:

entry 0x00100000 (phys)

etext 0x001032db (phys)

edata 0x0010ea16 (phys)

end 0x0010fd20 (phys)

Kernel executable memory footprint: 64KB

ebp = 0x00007b08 eip = 0x001009a7

arg0 = 0x00010094 arg1 = 0x00000000

arg2 = 0x00007b38 arg3 = 0x00100092

kern/debug/kdebug.c:307: print\_stackframe+22

ebp = 0x00007b18 eip = 0x00100c9a

arg0 = 0x00000000 arg1 = 0x00000000

arg2 = 0x00000000 arg3 = 0x00007b88

kern/debug/kmonitor.c:125: mon\_backtrace+10

ebp = 0x00007b38 eip = 0x00100092

arg0 = 0x00000000 arg1 = 0x00007b60

arg2 = 0xffff0000 arg3 = 0x00007b64

kern/init/init.c:48: grade\_backtrace2+33

ebp = 0x00007b58 eip = 0x001000bb

arg0 = 0x00000000 arg1 = 0xffff0000

arg2 = 0x00007b84 arg3 = 0x00000029

kern/init/init.c:53: grade\_backtrace1+38

ebp = 0x00007b78 eip = 0x001000d9

arg0 = 0x00000000 arg1 = 0x00100000

arg2 = 0xffff0000 arg3 = 0x0000001d

kern/init/init.c:58: grade\_backtrace0+23

ebp = 0x00007b98 eip = 0x001000fe

arg0 = 0x001032fc arg1 = 0x001032e0

arg2 = 0x0000130a arg3 = 0x00000000

kern/init/init.c:63: grade\_backtrace+34

ebp = 0x00007bc8 eip = 0x00100055

arg0 = 0x00000000 arg1 = 0x00000000

arg2 = 0x00000000 arg3 = 0x00010094

kern/init/init.c:28: kern\_init+84

ebp = 0x00007bf8 eip = 0x00007d68

arg0 = 0xc031fcfa arg1 = 0xc08ed88e

arg2 = 0x64e4d08e arg3 = 0xfa7502a8

<unknown>: -- 0x00007d67 --

++ setup timer interrupts

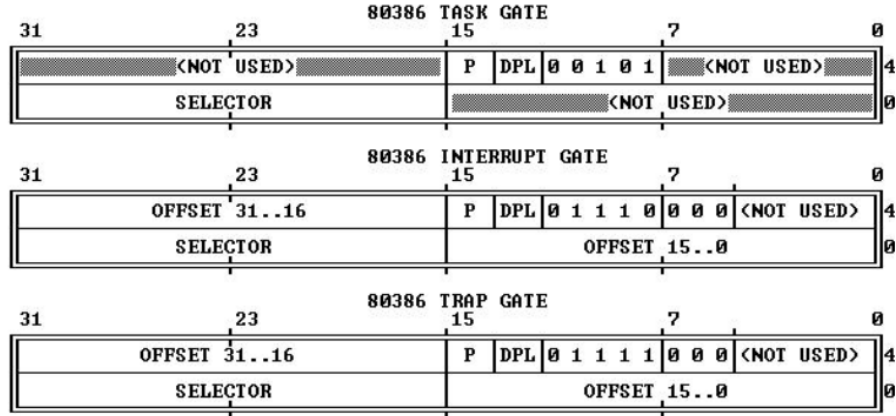
mijialong\$>

## 练习6 完善中断初始化和处理

- Interrupt-gate descriptor （中断方式用到）
- Trap-gate descriptor （系统调用用到）
- Task-gate descriptor （这里没有使用，不用考虑）

下图图显示了 80386 的任务门描述符、中断门描述符、陷阱门描述符的格式：

Figure 9-3. 80386 IDT Gate Descriptors



### 中断向量表中一个表项占多少字节

通过上图可以得到为8字节（64位）

### 其中哪几位代表中断处理代码的入口

通过上图可以知道

- 31 - 16位为中断例程的段选择子
- 15 - 0位和 63 - 48位为偏移量，分别是低位和高位

上面的数据决定中断代码的入口地址

### 完善 trap.c 中的 idt\_init()

查看 `mmu.h`，了解到 `SETGATE` 宏的作用和参数

```
/* *
 * Set up a normal interrupt/trap gate descriptor
 * - istrap: 1 代表陷阱门，0代表中断门
 * - sel: 中断/陷阱处理的段选择子
 * - off: 代码段的偏移量
 * - dpl: 特权级
 */
#define SETGATE(gate, istrap, sel, off, dpl) { \
    (gate).gd_off_15_0 = (uint32_t)(off) & 0xffff; \
    (gate).gd_ss = (sel); \
    (gate).gd_args = 0; \
    (gate).gd_rsv1 = 0; \
    (gate).gd_type = (istrap) ? STS_TG32 : STS_IG32; \
    (gate).gd_s = 0; \
    (gate).gd_dpl = (dpl); \
    (gate).gd_p = 1; \
    (gate).gd_off_31_16 = (uint32_t)(off) >> 16; \
}
```

```

void
idt_init(void)
{
    extern uintptr_t __vectors[];           // 声明 vector
    int i = 0;
    for (i = 0; i < 256; i++)               // 对于每一个表项进行
设置
    {
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
    lidt(&idt_pd);                          // 加载 idt
}

```

和同学讨论和查阅参考资料后，认为需要在该函数中实现用户态权限到内核态权限的切换，因此修改代码

```

void
idt_init(void)
{
    extern uintptr_t __vectors[];           // 声明 vector
    int i = 0;
    for (i = 0; i < 256; i++)               // 对于每一个表项进行设
置
    {
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
    用户态到内核态的权限切换
    SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, __vectors[T_SWITCH_TOK], DPL_USER);
    lidt(&idt_pd);                          // 加载 idt
}

```

## 完善 trap.c 中的 trap()

修改 tarp() 中的 trap\_dispatch()

```

trap_dispatch(struct trapframe *tf) {
    char c;

    switch (tf->tf_trapno) {
    case IRQ_OFFSET + IRQ_TIMER:
        {
            ticks++;
            if (ticks == TICK_NUM)
            {
                ticks = 0;
                print_ticks();
            }
        }
        break;
    .....
    }
}

```

编译后进行调试，如下图

```
serial [032]
100 ticks
serial [097] a
serial [119] w
serial [108] l
serial [115] s
serial [102] f
serial [104] h
serial [108] l
serial [097] a
serial [106] j
serial [102] f
serial [108] l
serial [097] a
serial [106] j
100 ticks
serial [115] s
serial [106] j
serial [103] g
serial [112] p
serial [106] j
serial [108] l
serial [120] x
serial [122] z
serial [110] n
serial [099] c
serial [108] l
serial [106] j
100 ticks
serial [112] p
serial [101] e
serial [097] a
serial [106] j
serial [108] l
serial [101] e
serial [110] n
serial [103] g
serial [108] l
serial [115] s
serial [106] j
serial [108] l
serial [059] ;
serial [100] d
serial [106] j
serial [115] s
100 ticks
serial [059] ;
serial [108] l
serial [003]
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```

## 实验结果



分析了 `bootmain.S` 代码，了解了 bootloader 进入保护模式的过程，以及加载 elf 格式的 os 过程

实现了函数调用堆栈跟踪函数，并且完善了中断初始化和处理

## 实验总结

### 问题1：尝试在 WSL 环境下运行相应代码

在 WSL 环境下，使用命令 `sudo apt install qemu` 安装 qemu 后，对 lab1 的代码进行编译，发现能够正常编译

```
root@LAPTOP-QTCGESHO:/mnt/d/ucore_os_lab/labcodes/lab1# make V=
+ cc kern/init/init.c
gcc -Ikern/lib/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/init/init.c -o obj/kern/init/init.o
kern/init/init.c:95:1: warning: 'lab1_switch_test' defined but not used [-Wunused-function]
lab1_switch_test(void) {
^
+ cc kern/lib/stdio.c
gcc -Ikern/lib/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/lib/stdio.c -o obj/kern/lib/stdio.o
+ cc kern/lib/readline.c
gcc -Ikern/lib/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/lib/readline.c -o obj/kern/lib/readline.o
+ cc kern/debug/panic.c
gcc -Ikern/debug/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/debug/panic.c -o obj/kern/debug/panic.o
kern/debug/panic.c: In function 'panic':
kern/debug/panic.c:27:5: warning: implicit declaration of function 'print_stackframe'; did you mean 'print_trapframe'? [-Wimplicit-function-declaration]
    print_stackframe();
    ^
+ cc kern/debug/kdebug.c
gcc -Ikern/debug/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/debug/kdebug.c -o obj/kern/debug/kdebug.o
kern/debug/kdebug.c:251:1: warning: 'read_eip' defined but not used [-Wunused-function]
read_eip(void) {
^
+ cc kern/debug/kmonitor.c
gcc -Ikern/debug/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/debug/kmonitor.c -o obj/kern/debug/kmonitor.o
+ cc kern/driver/clock.c
gcc -Ikern/driver/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/driver/clock.c -o obj/kern/driver/clock.o
+ cc kern/driver/console.c
gcc -Ikern/driver/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/driver/console.c -o obj/kern/driver/console.o
+ cc kern/driver/picirq.c
gcc -Ikern/driver/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/driver/picirq.c -o obj/kern/driver/picirq.o
+ cc kern/driver/intr.c
gcc -Ikern/driver/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/driver/intr.c -o obj/kern/driver/intr.o
+ cc kern/trap/trap.c
gcc -Ikern/trap/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/trap/trap.c -o obj/kern/trap/trap.o
kern/trap/trap.c:30:26: warning: 'idt_pd' defined but not used [-Wunused-variable]
static struct pseudotdesc idt_pd = {
^
+ cc kern/trap/trap.c
kern/trap/trap.c:14:13: warning: 'print_ticks' defined but not used [-Wunused-function]
static void print_ticks() {
^
+ cc kern/trap/vectors.S
gcc -Ikern/trap/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/trap/vectors.S -o obj/kern/trap/vectors.o
+ cc kern/trap/trapentry.S
gcc -Ikern/trap/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/trap/trapentry.S -o obj/kern/trap/trapentry.o
+ cc kern/mm/pmm.c
gcc -Ikern/mm/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Ikern/debug/ -Ikern/driver/ -Ikern/trap/ -Ikern/mm/ -c kern/mm/pmm.c -o obj/kern/mm/pmm.o
+ cc libs/string.c
gcc -Ilibs/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -c libs/string.c -o obj/libs/string.o
+ cc libs/printfat.c
gcc -Ilibs/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -c libs/printfat.c -o obj/libs/printfat.o
+ ld bin/kernel
ld -m elf_i386 -nostdlib -T tools/kernel.ld -o bin/kernel obj/kern/init/init.o obj/kern/lib/stdio.o obj/kern/lib/readline.o obj/kern/debug/panic.o obj/kern/debug/kdebug.o obj/kern/debug/kmonitor.o obj/kern/driver/clock.o obj/kern/driver/console.o obj/kern/driver/picirq.o obj/kern/driver/intr.o obj/kern/trap/trap.o obj/kern/trap/vectors.o obj/kern/trap/trapentry.o obj/kern/mm/pmm.o obj/libs/string.o obj/libs/printfat.o
+ cc boot/bootasm.S
gcc -Iboot/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootasm.S -o obj/boot/bootasm.o
+ cc boot/bootmain.c
gcc -Iboot/ -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc -fno-stack-protector -Ilibs/ -Os -nostdinc -c boot/bootmain.c -o obj/boot/bootmain.o
+ cc tools/sign.c
gcc -Itools/ -g -Wall -O2 -c tools/sign.c -o obj/sign/tools/sign.o
gcc -g -Wall -O2 obj/sign/tools/sign.o -o bin/sign
+ ld bin/bootblock
ld -m elf_i386 -nostdlib -N -e start -Ttext 0x7000 obj/boot/bootasm.o obj/boot/bootmain.o -o obj/bootblock.o
obj/bootblock.out: size 469 bytes
build 512 bytes boot sector: 'bin/bootblock' success!
dd if=/dev/zero of=bin/ucore.img count=10000
10000+0 records in
10000+0 records out
512000 bytes (512 KB, 4.9 MiB) copied, 0.078892 s, 65.6 MB/s
dd if=bin/bootblock of=bin/ucore.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.0172258 s, 29.6 KB/s
dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
146+1 records in
146+1 records out
74816 bytes (75 KB, 73 KiB) copied, 0.0093604 s, 8.0 MB/s
root@LAPTOP-QTCGESHO:/mnt/d/ucore_os_lab/labcodes/lab1#
```

使用 `make qemu V=` 则会报错

```
root@LAPTOP-QTCGESHO:/mnt/d/ucore_os_lab/labcodes/lab1# make qemu V=
qemu-system-i386 -no-reboot -parallel stdio -hda bin/ucore.img -serial null
WARNING: Image format was not specified for 'bin/ucore.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
Could not initialize SDL(No available video device) - exiting
Makefile:213: recipe for target 'qemu' failed
make: *** [qemu] Error 1
root@LAPTOP-QTCGESHO:/mnt/d/ucore_os_lab/labcodes/lab1#
```

其中包含了两个错误，第一个是关于镜像格式的错误；第二个是关于显示器设备的问题，可以通过添加参数 `-nographic` 来解决

使用命令 `make qemu-nox V=` 后发现能够得到想要的结果



```

make: Nothing to be done for 'TARGETS'.
root@LAPTOP-QTCGESHO:/mnt/d/ucore_os_lab/labcodes/lab1# make qemu-nox V=
qemu-system-i386 -no-reboot -serial mon:stdio -hda bin/ucore.img -nographic
WARNING: Image format was not specified for 'bin/ucore.img' and probing guessed raw.
        Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
        Specify the 'raw' format explicitly to remove the restrictions.
(THU.CST) os is loading ...

Special kernel symbols:
  entry 0x00100000 (phys)
  etext 0x001031f1 (phys)
  edata 0x0010ea16 (phys)
  end   0x0010fd20 (phys)
Kernel executable memory footprint: 64KB
++ setup timer interrupts
root@LAPTOP-QTCGESHO:/mnt/d/ucore_os_lab/labcodes/lab1#

```

这两条命令差别除了是否显示图形化界面之外，还包括了并行/串行端口的分配

接下来使用前一份实验报告中的 qemu 命令，尝试复现实验2第1部分

在此之前需要对命令进行修改，修改为 `qemu-system-i386 -S -s -d in_asm -D bin/q.log -serial mon:stdio -nographic -hda bin/ucore.img`

发现终端卡死，即使使用 `ctrl + c` 进行终止也没有反应

```

root@LAPTOP-QTCGESHO:/mnt/d/ucore_os_lab/labcodes/lab1# qemu-system-i386 -S -s -d in_asm -D bin/q.log -serial mon:stdio -nographic -hda bin/ucore.img
WARNING: Image format was not specified for 'bin/ucore.img' and probing guessed raw.
        Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
        Specify the 'raw' format explicitly to remove the restrictions.

```

但是在 gdb 进行调试之后，仍然会进入无限循环，原因是没有加上 `-no-reboot` 参数，使得 qemu 在程序执行完后会不断重启，从而出现了



```

Detaching from program: /mnt/d/ucores_lab/labcodes/lab1/bin/kernel, Remote target
Ending remote debugging.
root@LAPTOP-QTCGESHO:/mnt/d/ucores_lab/labcodes/lab1# qemu-system-i386 -S -s -d in_asm -D bin/q.log -serial mon:stdio -nographic -hda bin/ucores.img -no-reboot
WARNING: Image format was not specified for 'bin/ucores.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
(THU.CST) os is loading ...

Special kernel symbols:
  entry 0x00100000 (phys)
  etext 0x001001f1 (phys)
  edata 0x00100a16 (phys)
  end   0x00100d26 (phys)
Kernel executable memory footprint: 64KB
++ setup timer interrupts
root@LAPTOP-QTCGESHO:/mnt/d/ucores_lab/labcodes/lab1#

root@LAPTOP-QTCGESHO:/mnt/d/blog# cd ../ucores_lab/labcodes/lab1
root@LAPTOP-QTCGESHO:/mnt/d/ucores_lab/labcodes/lab1# gdb
GNU gdb (Ubuntu 8.1-0ubuntu2.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) file bin/kernel
Reading symbols from bin/kernel...done.
(gdb) target remote :1234
Remote debugging using :1234
0x00000000 in ?? ()
(gdb) c
Continuing.
Remote connection closed
(gdb) q
root@LAPTOP-QTCGESHO:/mnt/d/ucores_lab/labcodes/lab1#

```

经过控制变量法测试，发现是 `-S` 参数的问题，在没有该参数时，qemu 能够正常运行，在带有该参数时，则会导致终端卡死，无法接收中断信号

已知该参数是在模拟启动时，将 CPU 冻结，可以通过在 qemu 界面输入 `c` 使得 CPU 停止冻结，开始执行相关文件

猜测是因为使用了 `-serial mon:stdio` 参数，使得 OS 的进程反映到当前的终端上，而 `-S` 参数的存在导致了反映到当前终端的 OS 没有办法接收到信号，而 qemu 本身被 OS 覆盖，也因此无法接收到终端信号的输入，从而使得整个终端卡死

于是将串行端口的参数修改为 `-serial null` 再执行命令，进过测试，可以发现能够正常使用，符合上述猜测

```

root@LAPTOP-QTCGESHO:/mnt/d/ucores_lab/labcodes/lab1# qemu-system-i386 -d in_asm -D bin/q.log -serial null -nographic -hda bin/ucores.img -no-reboot -s
WARNING: Image format was not specified for 'bin/ucores.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
QEMU 2.11.1 monitor - type 'help' for more information
root@LAPTOP-QTCGESHO:/mnt/d/ucores_lab/labcodes/lab1# qemu-system-i386 -d in_asm -D bin/q.log -serial null -nographic -hda bin/ucores.img -no-reboot -s -S
WARNING: Image format was not specified for 'bin/ucores.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
QEMU 2.11.1 monitor - type 'help' for more information
(qemu) c
root@LAPTOP-QTCGESHO:/mnt/d/ucores_lab/labcodes/lab1#

```

综上，可以尝试使用 WSL1 来进行相关的模拟

但是由于 gcc 版本过高的问题，对于会遇到什么样的奇怪 bug 没有定数，因此暂时依然选择使用 moocos 虚拟机进行相关的练习，但是会在练习之后尝试使用 wsl 复现练习情况