



# 本科生实验报告

实验课程：\_\_\_\_\_操作系统\_\_\_\_\_

实验名称：\_\_\_\_\_保护模式\_\_\_\_\_

专业名称：\_\_\_\_\_计算机科学与技术\_\_\_\_\_

学生姓名：\_\_\_\_\_凌国明\_\_\_\_\_

学生学号：\_\_\_\_\_21307077\_\_\_\_\_

实验地点：\_\_\_\_\_教室\_\_\_\_\_

实验成绩：\_\_\_\_\_

报告时间：\_\_\_\_\_2023. 04. 20\_\_\_\_\_

## 1. 实验要求

- 学习 C 代码是如何通过预编译、编译、汇编和链接生成最终的可执行文件
- 学习一种 C/C++ 项目管理方案
- 学习 C 和汇编混合编程方法，即如何在 C 代码中调用汇编代码编写的函数和如何在汇编代码中调用使用 C 编写的函数
- 学习保护模式下的中断处理机制和可编程中断部件 8259A 芯片
- 通过编写实时钟中断处理函数来将本章的所有内容串联起来。

## 实验任务

- 复现 Example 1，结合具体的代码说明 C 代码调用汇编函数的语法和汇编代码调用 C 函数的语法。结合代码说明 `global`、`extern` 关键字的作用，结合代码说明 `global`、`extern` 关键字的作用
- 复现 Example 2，在进入 `setup_kernel` 函数后，将输出 `Hello World` 改为输出你的学号，结果截图并说说你是怎么做的
- 复现 Example 3，你可以更改 Example 中默认的中断处理函数为你编写的函数，然后触发之，结果截图并说说你是怎么做的
- 复现 Example 4，仿照 Example 中使用 C 语言来实现时钟中断的例子，利用 C/C++、`InterruptManager`、`STDIO` 和你自己封装的类来实现你的时钟中断处理过程，结果截图并说说你是怎么做的。注意，不可以使用纯汇编的方式来实现。（例如，通过时钟中断，你可以在屏幕的第一行实现一个跑马灯。跑马灯显示自己学号和英文名，即类似于 LED 屏幕显示的效果。）

## 2. 实验过程

### 1) 复现 example1

- 第一步：编写各个文件

```
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment1$ gedit c_func.c
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment1$ gedit cpp_func.cpp
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment1$ gedit asm_utils.asm
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment1$ gedit main.cpp
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment1$ ls
asm_utils.asm  c_func.c  cpp_func.cpp  main.cpp
```

- 第二步：编译与汇编，c 文件用 gcc，cpp 文件用 g++，asm 文件用 nasm

```
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment1$ gcc -o c_func.o -m32 -c c_func.c
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment1$ g++ -o cpp_func.o -m32 -c cpp_func.cpp
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment1$ g++ -o main.o -m32 -c main.cpp
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment1$ nasm -o asm_utils.o -f elf32 asm_utils.asm
```

- 第三步，链接

```
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment1$ g++ -o main.out main.o c_func.o cpp_func.o asm_utils.o -m32
```

- 第四步：执行

```
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment1$ ./main.out
Call function from assembly.
This is a function from C.
This is a function from C++.
Done.
```

- 第一步：编写 Makefile

```
~/os_lab/lab4/assignment1$ gedit Makefile

Makefile
~/os_lab/lab4/assignment1

main.out:main.o c_func.o cpp_func.o asm_utils.o
    g++ -o main.out main.o c_func.o cpp_func.o asm_utils.o -m32

main.o:main.cpp
    g++ -o main.o -c main.cpp -m32

c_func.o:c_func.c
    gcc -o c_func.o -c c_func.c -m32

cpp_func.o:cpp_func.cpp
    g++ -o cpp_func.o -c cpp_func.cpp -m32

asm_utils.o:asm_utils.asm
    nasm -o asm_utils.o asm_utils.asm -f elf32
```

- 第二步: make

```
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment1$ gedit Makefile
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment1$ ls
asm_utils.asm  c_func.c  cpp_func.cpp  main.cpp  Makefile
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment1$ make
g++ -o main.o -c main.cpp -m32
gcc -o c_func.o -c c_func.c -m32
g++ -o cpp_func.o -c cpp_func.cpp -m32
nasm -o asm_utils.o asm_utils.asm -f elf32
g++ -o main.out main.o c_func.o cpp_func.o asm_utils.o -m32
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment1$ ls
asm_utils.asm  c_func.c  cpp_func.cpp  main.cpp  main.out
asm_utils.o    c_func.o  cpp_func.o    main.o    Makefile
```

- 第三步: 运行

```
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment1$ ./main.out
Call function from assembly.
This is a function from C.
This is a function from C++.
Done.
```

关键字: 汇编代码中“global”关键字的作用是“使汇编符号对链接器可见”, 这意味着和这份汇编代码链接的所有代码都“认识”这个符号, 也就是可以“call”这个符号。C 代码中的“extern”关键字表示这里引用了一个文件外部的变量, 使用“extern”关键字可以使变量变为全局变量, 可以跨文件访问。

Make: 编写 makefile 时, 注意 C 文件用 gcc, C++ 文件用 g++, asm 汇编代码文件用 nasm。注意 gcc、g++ 中的 -m32 与 nasm 中的 -f elf32

## 2) 复现 example2

- 第一步：编写各项文件

```
asm_hello_world:
    push eax
    xor eax, eax
    ;mov ax, 0xb800
    ;mov gs, ax
    mov ah, 0x07
    mov al, '2'
    mov [gs:2 * (80 * 2 + 0)], ax

    mov al, '1'
    mov [gs:2 * (80 * 2 + 1)], ax

    mov al, '3'
    mov [gs:2 * (80 * 2 + 2)], ax

    mov al, '0'
    mov [gs:2 * (80 * 2 + 3)], ax

    mov al, '7'
    mov [gs:2 * (80 * 2 + 4)], ax

    mov al, '0'
    mov [gs:2 * (80 * 2 + 5)], ax

    mov al, '7'
    mov [gs:2 * (80 * 2 + 6)], ax

    mov al, '7'
    mov [gs:2 * (80 * 2 + 7)], ax

    pop eax
```

- 初始项目结构如下

```
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment2$ tree
.
├── build
├── include
│   ├── asm_utils.h
│   └── boot.inc
├── run
│   └── hd.img
└── src
    ├── boot
    │   ├── bootloader.asm
    │   ├── entry.asm
    │   └── mbr.asm
    ├── kernel
    │   └── setup.cpp
    └── utils
        └── asm_utils.asm
```

- 第二步：在 build 文件夹下编译 mbr, bootloader

```
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment2/build$ nasm -o mbr.bin -f bin -I../include/ ../src/boot/mbr.asm
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment2/build$ nasm -o bootloader.bin -f bin -I../include/ ../src/boot/bootloader.asm
```

- 第三步：编译内核代码

```
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment2/build$ nasm -o entry.obj  
-f elf32 ../src/boot/entry.asm  
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment2/build$ nasm -o asm_utils  
.o -f elf32 ../src/utils/asm_utils.asm  
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment2/build$ g++ -g -Wall -mar  
ch=i386 -m32 -nostdlib -fno-builtin -ffreestanding -fno-pic -I../include -c ../s  
rc/kernel/setup.cpp
```

- 第四步：链接生成两个可重定位文件，kernel.bin 只包含代码，kernel.o 是可执行文件


```
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment2/build$ ld -o kernel.o -m  
elf_i386 -N entry.obj setup.o asm_utils.o -e enter_kernel -Ttext 0x00020000  
ld: 警告：无法找到项目符号 enter_kernel; 缺省为 00000000000020000  
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment2/build$ objcopy -O binary  
kernel.o kernel.bin
```

- 第五步：将 mbr, bootloader, kernel 写入磁盘

```
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment2/build$ dd if=mbr.bin of=  
../run/hd.img bs=512 count=1 seek=0 conv=notrunc  
记录了1+0 的读入  
记录了1+0 的写出  
512 bytes copied, 0.000215107 s, 2.4 MB/s  
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment2/build$ dd if=bootloader.  
bin of=../run/hd.img bs=512 count=5 seek=1 conv=notrunc  
记录了0+1 的读入  
记录了0+1 的写出  
395 bytes copied, 0.000170045 s, 2.3 MB/s  
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment2/build$ dd if=kernel.bin  
of=../run/hd.img bs=512 count=200 seek=6 conv=notrunc  
记录了0+1 的读入  
记录了0+1 的写出  
164 bytes copied, 0.000167059 s, 982 kB/s
```

- 第六步：启动 qemu 测试

```
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment2$ cd run  
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment2/run$ qemu-system-i386 -h  
da hd.img -serial null -parallel stdio
```

A screenshot of a QEMU window titled "QEMU". The window shows the boot process of a virtual machine. The text inside the window includes "run bootloaderon 1.10.2-1ubuntu1)", "enter protect mode", and "21307077". The window also shows a warning message: "Warning: Detected probing guessed raw. s for raw images, write o".

```
run bootloaderon 1.10.2-1ubuntu1)  
enter protect mode  
21307077  
iPXE (http://ipxe.org) 00:03.0 C98
```

### 3) 复现 example3

- 第一步：编写各个代码文件

```
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment3$ gedit src/kernel/interr
upt.cpp

interrupt.cpp
~/os_lab/lab4/assignment3/src/kernel

#include "interrupt.h"
#include "os_type.h"
#include "os_constant.h"
#include "asm_utils.h"

InterruptManager::InterruptManager()
{
    // 构造函数
    initialize();
}

void InterruptManager::initialize()
{
    // 初始化IDT
    IDT = (uint32 *)IDT_START_ADDRESS;
```

构造函数，初始化函数，加载 idt 函数

```
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment3$ gedit src/utils/asm_util
s.asm

*asm_utils.asm
~/os_lab/lab4/assignment3/src/utils

[bits 32]
global asm_lidt
global asm_unhandled_interrupt
global asm_halt
```

asm\_unhandled\_interrupt 函数输出字符串

asm\_halt 函数死循环，asm\_lidt 函数加载 idt

```
load_kernel:
    push eax
    push ebx
    call asm_read_hard_disk ; 读取硬盘
    add esp, 8
    inc eax
    add ebx, 512
    loop load_kernel
```

```
jmp dword CODE_SELECTOR:KERNEL_START_ADDRESS ; 跳转到kernel
```

```
jmp $ ; 死循环
```

Bootloader 最后要读 200 个扇区，最后跳转到 kernel

编写文件过程中要特别注意各个 include 关系

- 第二步：复制教程中的 makefile

```
ASM_COMPILER = nasm
C_COMPILER = gcc
CXX_COMPILER = g++
CXX_COMPILER_FLAGS = -g -Wall -march=i386 -m32 -nostdlib -fno-builtin -ffreestanding -fno-pic
LINKER = ld

SRCDIR = ../src
RUNDIR = ../run
BUILDDIR = build
INCLUDE_PATH = ../include

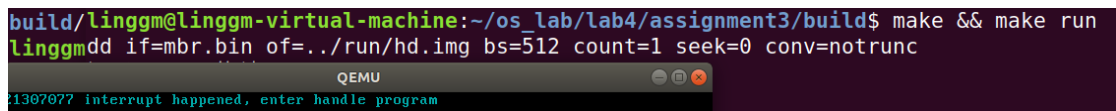
CXX_SOURCE += $(wildcard $(SRCDIR)/kernel/*.cpp)
CXX_OBJ += $(CXX_SOURCE:$(SRCDIR)/kernel/%.cpp=%.o)

ASM_SOURCE += $(wildcard $(SRCDIR)/utils/*.asm)
ASM_OBJ += $(ASM_SOURCE:$(SRCDIR)/utils/%.asm=%.o)

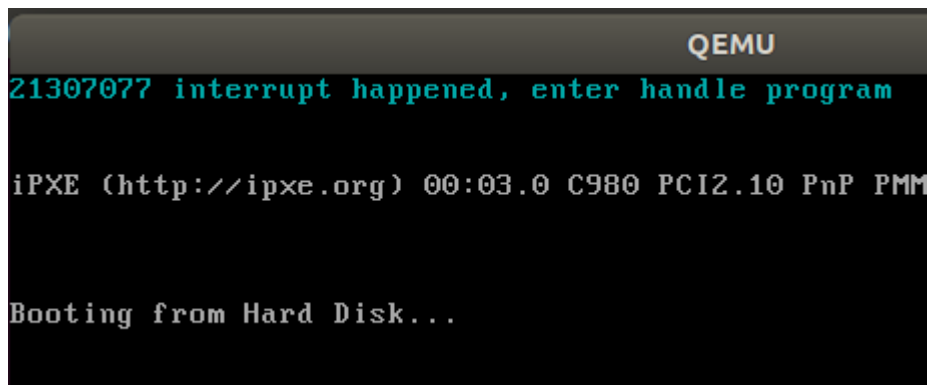
OBJ += $(CXX_OBJ)
OBJ += $(ASM_OBJ)

build : mbr.bin bootloader.bin kernel.bin kernel.o
    dd if=mbr.bin of=$(RUNDIR)/hd.img bs=512 count=1 seek=0 conv=notrunc
    dd if=bootloader.bin of=$(RUNDIR)/hd.img bs=512 count=5 seek=1 conv=notrunc
    dd if=kernel.bin of=$(RUNDIR)/hd.img bs=512 count=145 seek=6 conv=notrunc
# nasm的include path有一个尾随/
```

- 因为ex2中已经实现了用nasm,gcc,g++编译汇编链接各个文件的过程，这里直接用makefile。make run效果如下



```
build/linggm@linggm-virtual-machine:~/os_lab/lab4/assignment3/build$ make && make run
linggmdd if=mbr.bin of=../run/hd.img bs=512 count=1 seek=0 conv=notrunc
QEMU
1307077 interrupt happened, enter handle program
```



```
QEMU
21307077 interrupt happened, enter handle program

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM

Booting from Hard Disk...
```



#### 4) 复现 example4

- 第一步，编写 stdio 类

```
class STDIO
{
private:
    uint8 *screen;

public:
    STDIO();
    // 初始化函数
    void initialize();
    // 打印字符c, 颜色color到位置(x,y)
    void print(uint x, uint y, uint8 c, uint8 color);
    // 打印字符c, 颜色color到光标位置
    void print(uint8 c, uint8 color);
    // 打印字符c, 颜色默认到光标位置
    void print(uint8 c);
    // 移动光标到一维位置
    void moveCursor(uint position);
    // 移动光标到二维位置
    void moveCursor(uint x, uint y);
    // 获取光标位置
    uint getCursor();

private:
    // 滚屏
    void rollUp();
}
```

类的声明如上

```
// 处理高8位
temp = (position >> 8) & 0xff;
asm_out_port(0x3d4, 0x0e);
asm_out_port(0x3d5, temp);

// 处理低8位
temp = position & 0xff;
asm_out_port(0x3d4, 0x0f);
asm_out_port(0x3d5, temp);
```

注意：在对光标位置进行读写时，temp 是八位的，pos 是十六位的，所以 temp=pos>>8 表示 temp 赋值为 pos 的高八位；而 temp=pos 进行了位截断，表示 pos 的低八位；然后对两个端口进行读写即可

- 第二步：编写 interrupt 类

```
public:
    InterruptManager();
    void initialize();
    // 设置中断描述符
    // index    第index个描述符, index=0, 1, ..., 255
    // address  中断处理程序的起始地址
    // DPL      中断描述符的特权级
    void setInterruptDescriptor(uint32 index, uint32 address, byte DPL);
    // 开启时钟中断
    void enableTimeInterrupt();
    // 禁止时钟中断
    void disableTimeInterrupt();
    // 设置时钟中断处理函数
    void setTimeInterrupt(void *handler);

private:
    // 初始化8259A芯片
    void initialize8259A();
```

类的声明如上

```
// 初始化中断计数变量
times = 0;

// 初始化IDT
IDT = (uint32 *)IDT_START_ADDRESS;
asm_lidt(IDT_START_ADDRESS, 256 * 8 - 1);

for (uint i = 0; i < 256; ++i)
{
    setInterruptDescriptor(i, (uint32)asm_unhandled_interrupt, 0);
}

// 初始化8259A芯片
initialize8259A();
```

相比 assignment3, interrupt 类的 init 函数新增了“初始化中断次数”和“初始化 8259A”芯片

```
// OCW 1 屏蔽主片所有中断, 但主片的IRQ2需要开启
asm_out_port(0x21, 0xfb);
// OCW 1 屏蔽从片所有中断
asm_out_port(0xa1, 0xff);
```

“初始化 8259A 芯片”的步骤较为固定, 不赘述, 但要注意屏蔽所有主从片中断 (因为没有建立 8259A 芯片的中断处理函数)

```

void InterruptManager::enableTimeInterrupt()
{
    uint8 value;
    // 读入主片ocw
    asm_in_port(0x21, &value);
    // 开启主片时钟中断, 置0开启
    value = value & 0xfe;
    asm_out_port(0x21, value);
}

void InterruptManager::disableTimeInterrupt()
{
    uint8 value;
    asm_in_port(0x21, &value);
    // 关闭时钟中断, 置1关闭
    value = value | 0x01;
    asm_out_port(0x21, value);
}

```

开关时钟中断部分：0x21 端口的第 0 位表示 IRQ0 的屏蔽，即时钟中断的屏蔽，将第 0 位置 0 表示屏蔽时钟中断

- 第三步：在 interrupt 类中编写自己的中断处理函数

```

// 中断处理函数
extern "C" void c_time_interrupt_handler()
{
    // 清空屏幕
    for (int i = 0; i < 80; ++i)
    {
        stdio.print(0, i, ' ', 0x07);
    }

    // 中断发生的次数++
    ++times;
    // 输出字符串
    char str[] = "21307077_lingguoming_clock_interrupt";

    // 移动光标到(0,0)输出字符
    stdio.moveCursor(0);
    for(int i = 0; str[i]; ++i ) {
        // 跑马灯
        if(i - times % 36 > -3 && i - times % 36 < 3)
            stdio.print(str[i], 0x03);
        else
            stdio.print(str[i]);
    }
}

```

实现了利用时钟中断，进行跑马灯输出学号和姓名的功能

- 第四步：编写 setup.cpp

```
extern "C" void setup_kernel()
{
    // 中断处理部件
    interruptManager.initialize();
    // 屏幕Io处理部件
    stdio.initialize();
    interruptManager.enableTimeInterrupt();
    interruptManager.setTimeInterrupt((void *)asm_time_interrupt_handler);
    asm_enable_interrupt();
    asm_halt();
}
```

进行 asm\_halt() 后，不断执行 jmp \$，每隔一段时间触发时钟中断，从而转向中断处理程序进行跑马灯输出

- 第五步：启动 qemu 运行，结果如下

```
~/os_lab/lab4/assignment4/build$ make && make run
QEMU
21307077_lingguoming_clock_interrupt
```

```
QEMU
21307077_lingguoming_clock_interrupt_
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10
Booting from Hard Disk...
```

```
QEMU
21307077_lingguoming_clock_interrupt_
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10
Booting from Hard Disk...
```

## 关键代码

### 1) Assignment1

所有代码与教程一致

### 2) Assignment2 (debug 信息见过程部分)

```
asm_hello_world:
    push eax
    xor eax, eax
    ;mov ax, 0xb800
    ;mov gs, ax
    mov ah, 0x07
    mov al, '2'
    mov [gs:2 * (80 * 2 + 0)], ax

    mov al, '1'
    mov [gs:2 * (80 * 2 + 1)], ax

    mov al, '3'
    mov [gs:2 * (80 * 2 + 2)], ax

    mov al, '0'
    mov [gs:2 * (80 * 2 + 3)], ax

    mov al, '7'
    mov [gs:2 * (80 * 2 + 4)], ax

    mov al, '0'
    mov [gs:2 * (80 * 2 + 5)], ax

    mov al, '7'
    mov [gs:2 * (80 * 2 + 6)], ax

    mov al, '7'
    mov [gs:2 * (80 * 2 + 7)], ax

    pop eax
```

进入 setup\_kernel 函数后，调用 asm\_hello\_world，将我的学号写入显存，输出到屏幕上。只有 asm\_hello\_world 代码与教程不同，所有仅展示这部分代码，其他省略。

### 3) Assignment3

```
load_kernel:
    push eax
    push ebx
    call asm_read_hard_disk ; 读取硬盘
    add esp, 8
    inc eax
    add ebx, 512
    loop load_kernel
```

```
jmp dword CODE_SELECTOR:KERNEL_START_ADDRESS ; 跳转到kernel
```

```
jmp $ ; 死循环
```

Bootloader 最后要读 200 个扇区，最后跳转到 kernel

#### 4) Assignment4

```
// OCW 1 屏蔽主片所有中断，但主片的IRQ2需要开启
asm_out_port(0x21, 0xfb);
// OCW 1 屏蔽从片所有中断
asm_out_port(0xa1, 0xff);
```

“初始化 8259A 芯片”的步骤较为固定，不赘述，但要注意屏蔽所有主从片中中断（因为没有建立 8259A 芯片的中断处理函数）

```
void InterruptManager::enableTimeInterrupt()
{
    uint8 value;
    // 读入主片OCW
    asm_in_port(0x21, &value);
    // 开启主片时钟中断，置0开启
    value = value & 0xfe;
    asm_out_port(0x21, value);
}
```

开关时钟中断部分：0x21 端口的第 0 位表示 IRQ0 的屏蔽，即时钟中断的屏蔽，将第 0 位置 0 表示屏蔽时钟中断

```
// 中断处理函数
extern "C" void c_time_interrupt_handler()
{
    // 清空屏幕
    for (int i = 0; i < 80; ++i)
    {
        stdio.print(0, i, ' ', 0x07);
    }

    // 中断发生的次数++
    ++times;
    // 输出字符串
    char str[] = "21307077_lingguoming_clock_interrupt";

    // 移动光标到(0,0)输出字符
    stdio.moveCursor(0);
    for(int i = 0; str[i]; ++i) {
        // 跑马灯
        if(i - times % 36 > -3 && i - times % 36 < 3)
            stdio.print(str[i], 0x03);
        else
            stdio.print(str[i]);
    }
}
```

定义字符串 str 为我的学号和姓名，然后 for 遍历字符串进行输出字符串的长度为 36，当中断次数 times % 36 和 字符串索引 i 距离较近时，对这部分字符进行着色（青色 0x03），其他字符则是默认颜色（白色 0x07）

### 3 实验结果

#### 1) Assignment1

```
linggm@linggm-virtual-machine:~/os_lab/lab4/assignment1$ ./main.out
Call function from assembly.
This is a function from C.
This is a function from C++.
Done.
```

#### 2) Assignment2

```
run bootloader on 1.10.2-1ubuntu1)
enter protect mode
21307077
iPXE (http://ipxe.org) 00:03.0 C980
```

#### 3) Assignment3

```
QEMU
21307077 interrupt happened, enter handle program

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM

Booting from Hard Disk...
```

#### 4) Assignment4

```
QEMU
21307077_lingguoming_clock_interrupt_

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10

Booting from Hard Disk...
```

```
QEMU
21307077_lingguoming_clock_interrupt_

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10

Booting from Hard Disk...
```