

实验四报告

个人信息

- 姓名：李浩辉
- 学号：21307018

实验要求

1. 复现example1，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。
2. 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。
3. 复现Example 3，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。
4. 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。

实验过程

Assignment 1

实验要求

混合编程复现

实验原理

C++中调用asm函数，asm函数内容为调用C语言，cpp中的函数

注意事项：

- 引用外部的参数都需要添加extern语句

```
extern function_from_C           ; asm引用c,cpp函数
extern function_from_CPP
extern void function_from_asm();  //C引用asm函数
extern "C" void function_from_asm(); //CPP引用asm函数
```

- 函数给外部引用需要作特别声明(cpp,asm)

```
global function_from_asm           ; asm函数给外部用
extern "C" void function_from_asm(); //cpp代码给外部用
```

- asm中调用cpp/c函数，如果有参数，**从右到左依次入栈，返回值放在eax**

```
int function_from_C(int arg1, int arg2);//这是C中声明
```

； 以下是asm中的调用

```
push 2          ; arg2
push 1          ; arg1
call function_from_C
add esp, 8      ; 清除栈上的参数
```

实验内容

三个function文件：c_function, cpp_function, asm_function

c/cpp函数均为输出语句；

asm函数则为调用c以及cpp的函数；

main函数则为调用asm函数。

因此，有几点：

- cpp文件中函数声明要加上extern "C"

```
extern "C" void function_from_CPP()
```

- asm函数声明要提前声明global

```
global function_from_asm
```

- asm引用c/cpp函数前要有extern 声明

```
extern function_from_C
extern function_from_CPP
```

- main函数（cpp）调用前要有声明

```
extern "C" void function_from_asm();
```

接着开始编译链接

4个文件都编译成可重定位文件即 .o 文件，接着链接成可执行文件 (.out)

```
gcc -o c_func.o -m32 -c c_func.c
g++ -o cpp_func.o -m32 -c cpp_func.cpp
g++ -o main.o -m32 -c main.cpp
nasm -o asm_utils.o -f elf32 asm_utils.asm
g++ -o main.out main.o c_func.o cpp_func.o asm_utils.o -m32
```

编译时g++用于cpp文件, gcc用于c语言, nasm用于asm文件
结果有如下输出

```
lihh@lihh-VirtualBox:~/lab4/example1$ make run
./main.out
Call function from assembly.
This is a function from C.
This is a function from C++.
Done.
```

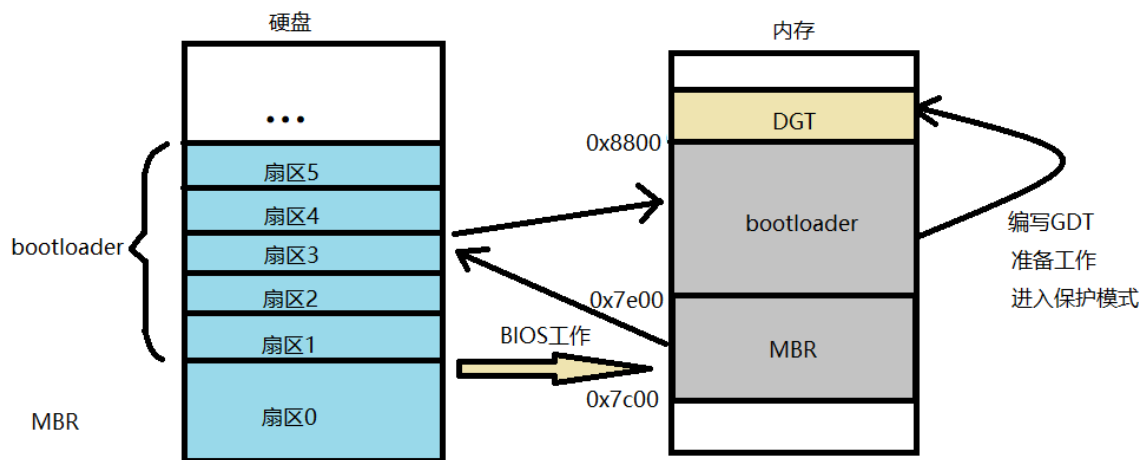
Assignment 2

实验要求

加载内核, 内核代码写为输出学号

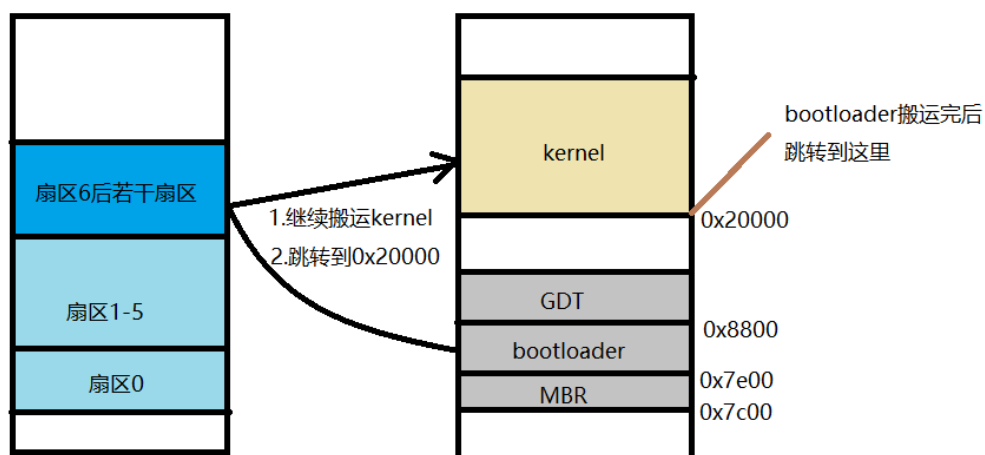
实验原理

上一次MBR, bootloader内容如下



这次bootloader 在这个前提下, 继续完成**加载内核**操作

after the above:



文件分类

- `build` 存放makefile以及一些.o和.bin文件
- `include` 存放.h等函数定义，常量定义的头文件
- `run` 存放磁盘映像以及gdbinit
- `src` 存放各种函数实现文件

说下**编译链接**的知识点：

- 分开文件夹放后，**我们include并不需要写文件路径**，因为在编译时我们makefile在/build中并且编译指令写上路径。
- **所有asm函数声明在/include/asm_utils.h**。要用只需要 `#include "asm_utils.h"`
- 内核有多部分文件组成，各种cpp,c,asm代码先生成.o文件，再用**ld指令链接**。**需要给出起始地址即** `-Ttext 0x00020000` 。
- 内核链接过程中，二进制文件的起始地址存放的并不一定是我们希望的内核进入地址。因此，我们需要**将内核进入点的代码放在ld的所有.o文件之首**。即

```
kernel.o : entry.obj $(OBJ)
$(LINKER) -o kernel.o -melf_i386 -N entry.obj $(OBJ) -e enter_kernel -Ttext 0x00020000
```

这里的entry.obj必须在\$（OBJ）之前。

- 再继续生成kernel.bin。
简要理解：bin文件写入磁盘，.o文件用于gdb调试。

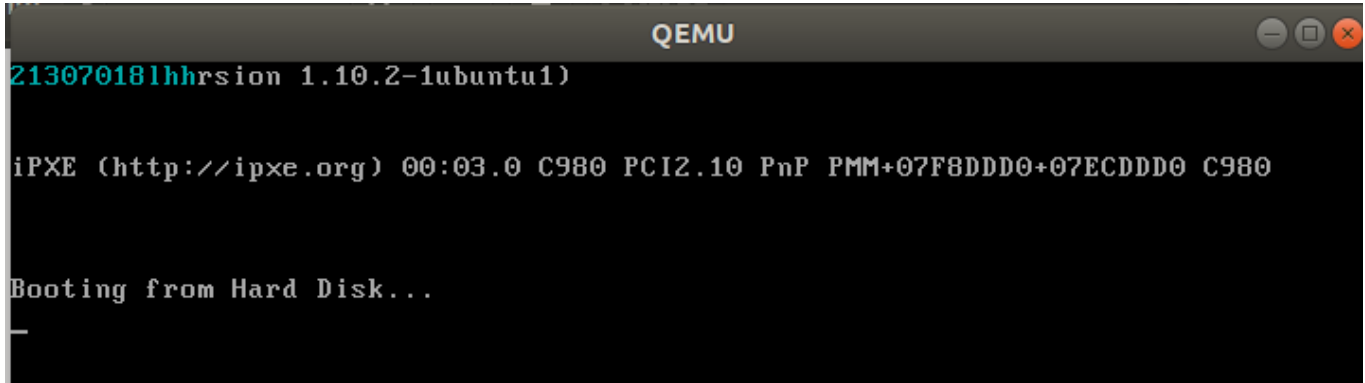
```
objcopy -O binary kernel.o kernel.bin
```

实验内容

完成一些准备工作，在kernel/setup.cpp中输出学号即可

如果有输出即代表加载并成功跳转入kernel

效果如图



Assignment 3

实验要求

写中断，触发中断进行测试

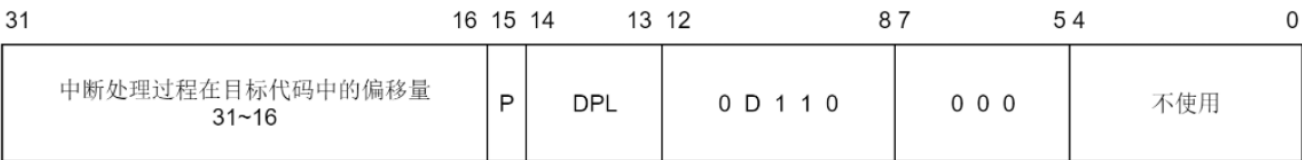
实验原理

- BIOS内置的中断程序是16位的。所以，在保护模式下这些代码便不再适用。
- **中断处理机制：**
 1. CPU检测到有中断信号
 2. 根据**中断向量号**到IDT取得对应的**中断描述符**
 3. 根据中断描述符中的**段选择子**到GDT中取**段描述符**
 4. CPU 根据特权级设定即将运行程序的栈地址
 5. 保护现场
 6. 跳转执行
 7. iret返回

信息流动过程

中断向量号- ——> 中断描述符- ——> 段描述符

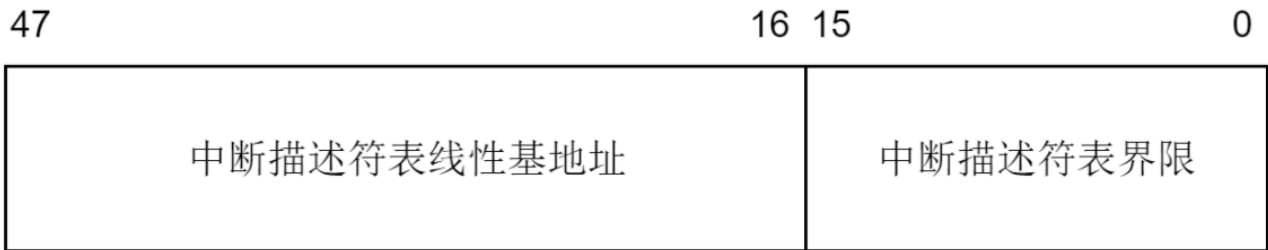
中断描述符内容



- 偏移量：中断程序的代码在中断程序所在段的偏移位置。
 - 段选择子：中断程序所在段的选择子。
 - P位：段存在位。 0表示不存在，1表示存在。
 - DPL：特权级描述。 0-3 共4级特权，特权级从0到3依次降低。
 - D位： D=1表示32位代码，D=0表示16位代码。
 - 保留位：保留不使用。
- 简要理解：段选择子放到CS寄存器，偏移量放到EIP可以到中断程序**

名称	位置	信息存放寄存器	存放内容
IDT	任意存放	IDTR	中断描述符
GDT	任意存放	GDTR	段描述符

IDTR内容类似与GDTR



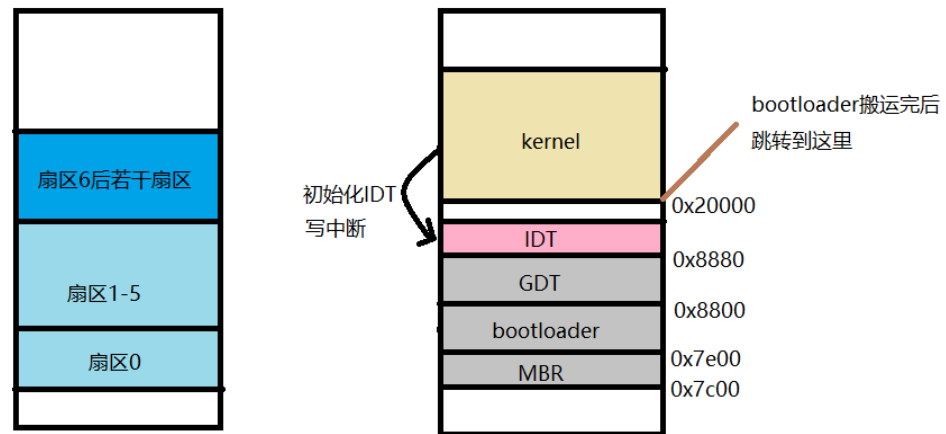
实验内容

工作内容

- 完成IDT的初始化
- 定义中断默认处理函数

- 初始化256个中断描述符

after the above:



实现步骤

以下步骤均在跳转到kernel后跳转到setup_kernel部分进行
提前创建一个InterruptManager类以进行操作

```
class InterruptManager
{
private:
    // IDT起始地址
    uint32 *IDT;

public:
    InterruptManager();
    // 初始化
    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);
};
```

初始化IDT

```

void InterruptManager::initialize()
{
    // 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_interrupt_empty_handler, 0);
    }
}

```

这里IDT地址已经提前设定好

```
#define IDT_START_ADDRESS 0x8880
```

设定IDTR

```
lidt [tag]
```

C语言中没有这个用法于是我们提前在asm准备好函数以供其使用。

```

; void asm_lidt(uint32 start, uint16 limit)
asm_lidt:
    push ebp
    mov ebp, esp
    push eax

    mov eax, [ebp + 4 * 3]
    mov [ASM_IDTR], ax
    mov eax, [ebp + 4 * 2]
    mov [ASM_IDTR + 2], eax
    lidt [ASM_IDTR]

    pop eax
    pop ebp
    ret

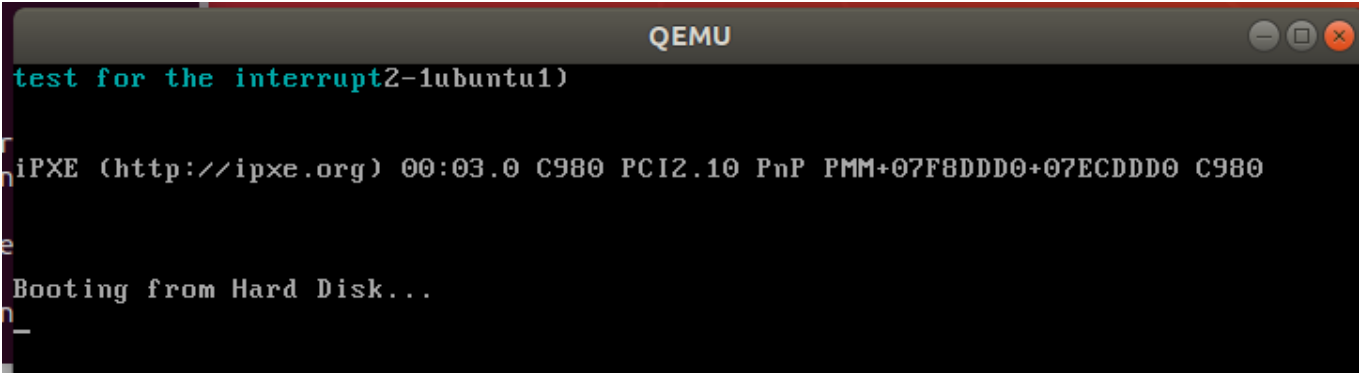
ASM_IDTR dw 0
         dd 0

```


我们设置的**所有中断段选择子和偏移量都一样**，于是都指向了同一部分代码
内容为输出语句“Unhandled interrupt happened, halt...”这一部分可以随便改

```
ASM_UNHANDLED_INTERRUPT_INFO db 'Unhandled interrupt happened, halt...'
                                db 0
; void asm_unhandled_interrupt()
asm_unhandled_interrupt:
    cli
    mov esi, ASM_UNHANDLED_INTERRUPT_INFO
    xor ebx, ebx
    mov ah, 0x03
.output_information:
    cmp byte[esi], 0
    je .end
    mov al, byte[esi]
    mov word[gs:bx], ax
    inc esi
    add ebx, 2
    jmp .output_information
.end:
    jmp $
```

我们把语句修改下，然后尝试触发。




尝试

尝试写一个新的中断，触发之。

保护模式下中断有默认规则
除0错误得到的中断向量号是0

向量号	助记符	说明	类型	错误号	产生源
0	#DE	除出错	故障	无	DIV或IDIV指令

首先我们可以在 `asm_utils_asm` 编写想要的asm中断函数

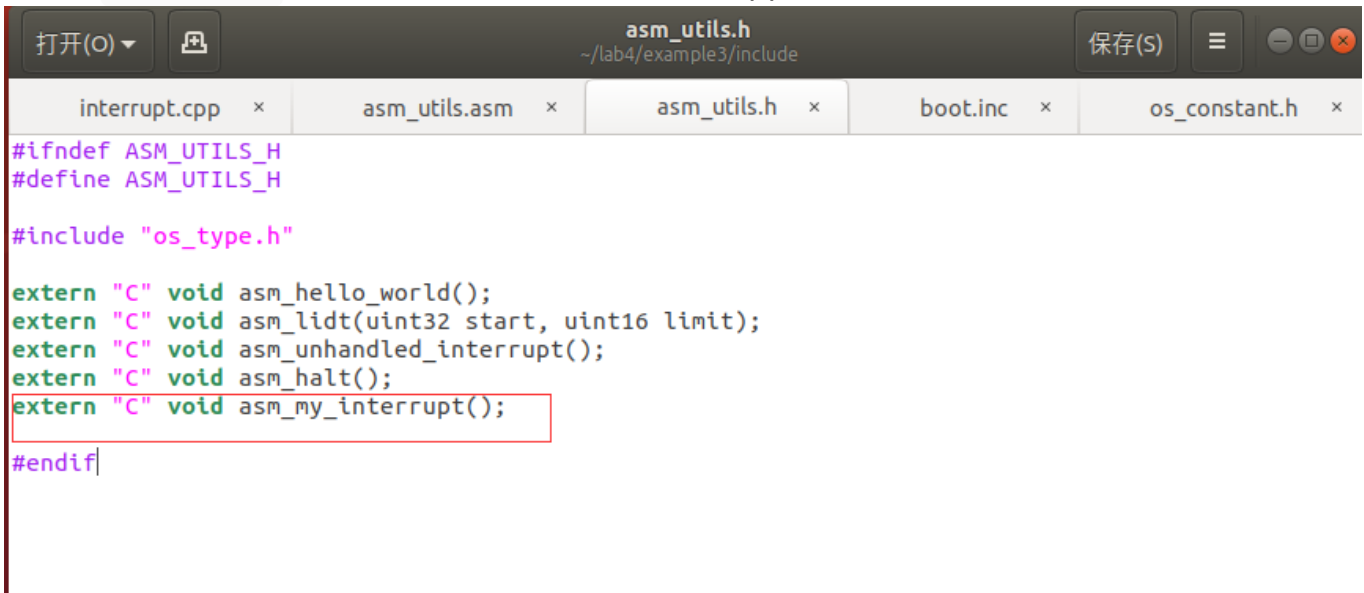


```
[bits 32]

global asm_hello_world
global asm_lidt
global asm_unhandled_interrupt
global asm_halt
global asm_my_interrupt

asm_my_interrupt:
    cli
    mov ah, 0x03
    mov al, 'l'
    mov [gs:2*3], ax
    mov al, 'h'
    mov [gs:2*4], ax
    mov [gs:2*5], ax
    jmp $
```

接着在 `asm_utils.h` 中声明该函数，这样我们就可以cpp中使用这个函数了



```
asm_utils.h
~/lab4/example3/include

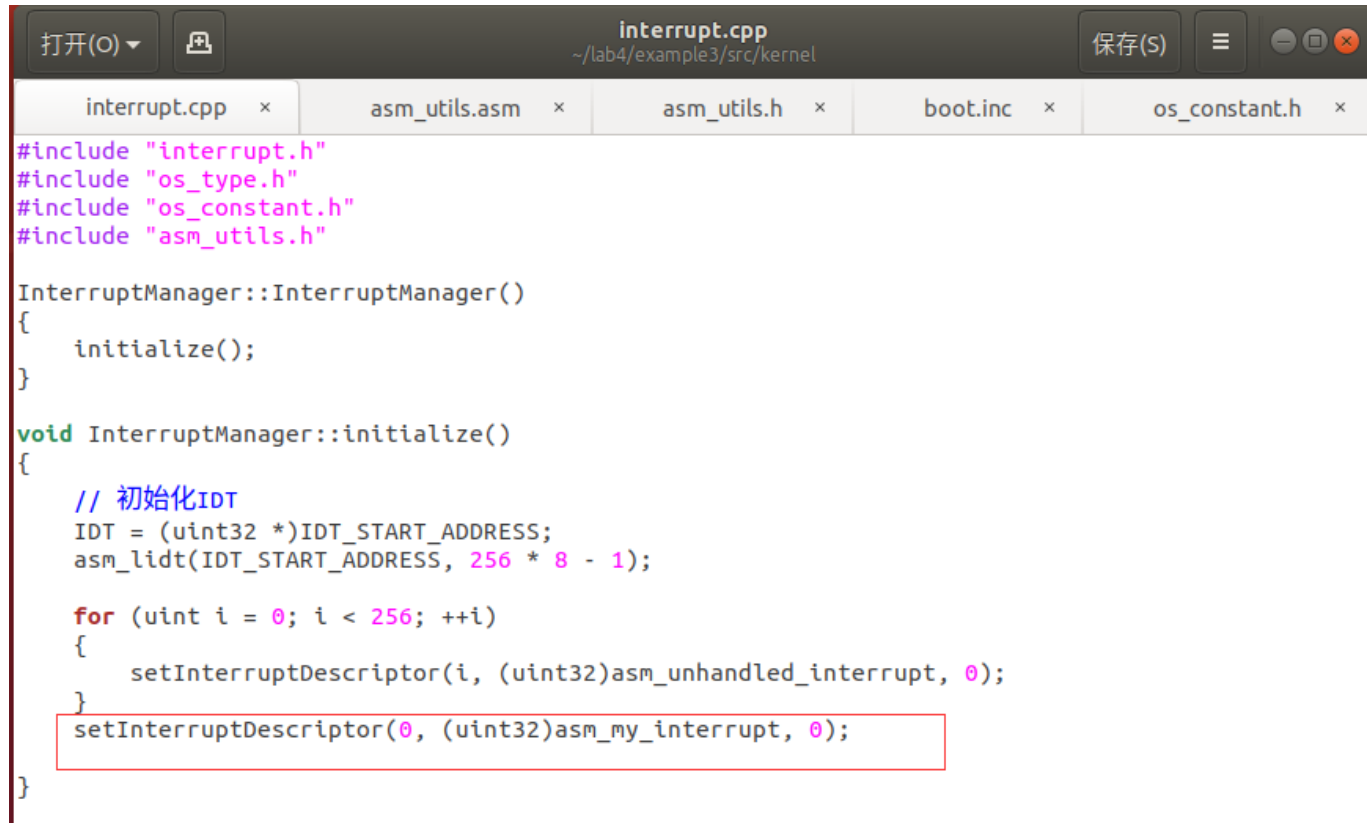
#ifndef ASM_UTILS_H
#define ASM_UTILS_H

#include "os_type.h"

extern "C" void asm_hello_world();
extern "C" void asm_lidt(uint32 start, uint16 limit);
extern "C" void asm_unhandled_interrupt();
extern "C" void asm_halt();
extern "C" void asm_my_interrupt();

#endif
```

把这个函数地址放进到向量号为0的IDT中，这样我们只要除以0就可以导向到这个函数。



```
interrupt.cpp x  asm_utils.asm x  asm_utils.h x  boot.inc x  os_constant.h x
#include "interrupt.h"
#include "os_type.h"
#include "os_constant.h"
#include "asm_utils.h"

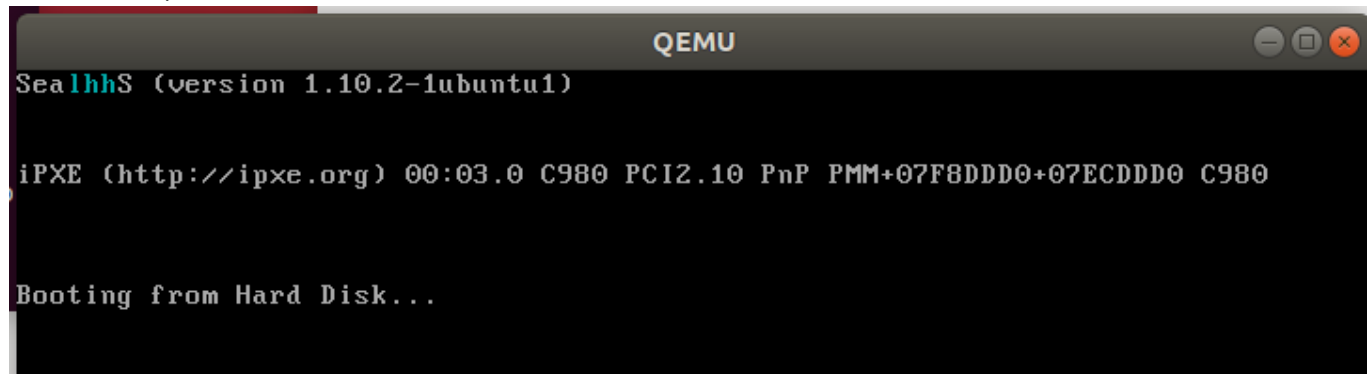
InterruptManager::InterruptManager()
{
    initialize();
}

void InterruptManager::initialize()
{
    // 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);
    }
    setInterruptDescriptor(0, (uint32)asm_my_interrupt, 0);
}
```

在setup的时候除以0故意触发这个中断。

再次make ,make run 就可以发现我们的中断设置成功了。



```
QEMU
SeaLhhS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD+07ECDDDD C980

Booting from Hard Disk...
```

Assignment 4

实验要求

使用C语言来实现时钟中断的例子

实验原理

8259A芯片

可编程中断控制器 (PIC: Programmable Interrupt Controller) , IRQ0-IRQ7, 默认优先级从高到低。这些信号线与外设相连, 外设通过IRQ向8259A芯片发送中断请求。

ICW

ICW1~ICW4 (初始化命令字, Initialization Command Words)

- **ICW1**

端口0x20,0xA0

7	6	5	4	3	2	1	0
0	0	0	1	M	0	C	I

- I位: 若置1, 表示ICW4会被发送。置0表示ICW4不会被发送。我们会发送ICW4, 所以I位置1。
- C位: 若置0, 表示8259A工作在级联环境下。8259A的主片和从片我们都会使用到, 所以C位置0。
- M位: 指出中断请求的电平触发模式, 在PC机中, M位应当被置0, 表示采用“边沿触发模式”。

- **ICW2**

端口0x21,0xA1

7	6	5	4	3	2	1	0
A7	A6	A5	A4	A3	0	0	0

对于主片和从片, ICW2都是用来表示当IRQ0的中断发生时, 8259A会向CPU提供的中断向量号。此后, IRQ0, IRQ1, ..., IRQ7的中断号为ICW2, ICW2+1, ICW2+2, ..., ICW+7。

- **ICW3**

端口0x21,0xA1

主片

0-外设 1-从片

7	6	5	4	3	2	1	0
IRQ7	IRQ6	IRQ5	IRQ4	IRQ3	IRQ2	IRQ1	IRQ0

从片

IRQ指示主片哪一个IRQ连接从片

7	6	5	4	3	2	1	0
0	0	0	0	0	IRQ		

- **ICW4**
端口0x21,0xA1

- EOI位：若置1表示自动结束，在PC位上这位需要被清零。
- 80×86位：置1表示PC工作在80x86架构下，因此我们置1

7	6	5	4	3	2	1	0
0	0	0	0	0	0	EOI	80x86

OCW

- **OCW1 中断屏蔽，发送到0x21（主片）或0xA1（从片）端口。**
1-屏蔽

7	6	5	4	3	2	1	0
IRQ7	IRQ6	IRQ5	IRQ4	IRQ3	IRQ2	IRQ1	IRQ0

- **OCW2 一般用于发送EOI消息，发送到0x20（主片）或0xA0（从片）端口。**
EOI信息发送0x20

对于8259A芯片产生的中断，我们需要手动在中断返回前向8259A发送EOI消息。如果没有发送EOI消息，那么此后的中断便不会被响应。

7	6	5	4	3	2	1	0
R	SL	EOI	0	0	L2	L1	L0

- **OCW3 用于设置下一个读端口动作将要读取的IRR或ISR**

名称	端口	作用
OCW1	0x21/0xA1	屏蔽
OCW2	0x20/0xA0	EOI消息
OCW3		设置IRR/ISR

时钟中断

在计算机中，有一个称为8253的芯片，其能够以一定的频率来产生时钟中断。当其产生了时钟中断后，信号会被8259A截获，从而产生IRQ0中断。

光标操作

屏幕的像素为25*80，所以光标的位置从上到下，从左到右依次编号为0-1999，用16位表示。

端口	内容	作用
0×3d4	0×0e/0×0f	表示高/低8位
0×3d5	光标位置	读/写

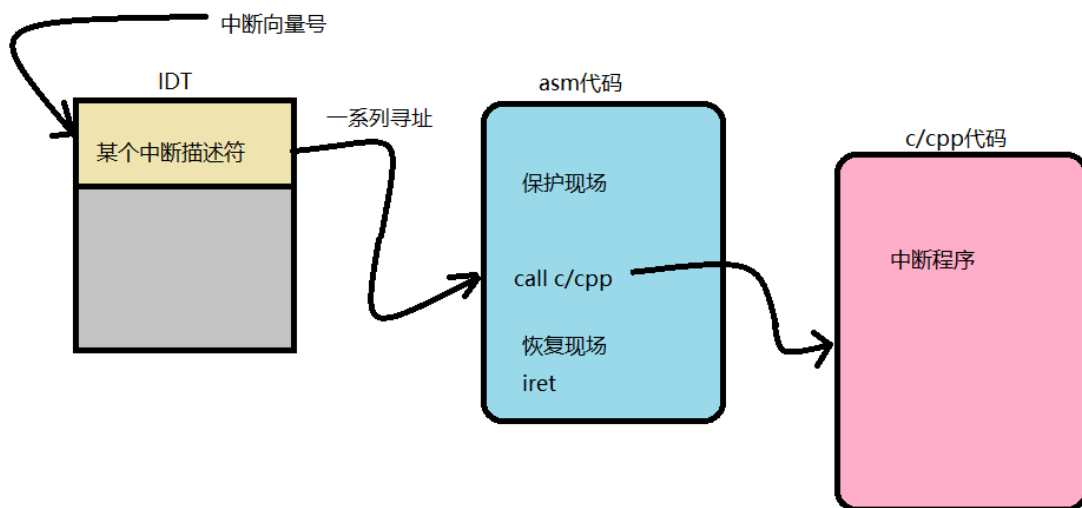
实验内容

- 编写中断处理函数
- 设置主片IRQ0中断对应的中断描述符
- 开启时钟中断
- 开中断

中断流程

- 保护现场
- 中断程序执行
- 恢复现场

C语言不具备相关指令（iret指令），于是我们只能先跳到汇编代码，保护现场，通过call跳转到C语言代码，C代码执行完返回到汇编，然后可以恢复现场，iret返回



主体内容如下代码注释所说

```

extern "C" void setup_kernel()
{
    // 中断处理部件，初始化IDT
    interruptManager.initialize();

    // 屏幕IO处理部件，设置好一些C语言函数
    stdio.initialize();

    //开时钟中断
    interruptManager.enableTimeInterrupt();

    //这里跳转到汇编代码，汇编中跳转到C，C返回到汇编继续完成
    interruptManager.setTimeInterrupt((void *)asm_time_interrupt_handler);

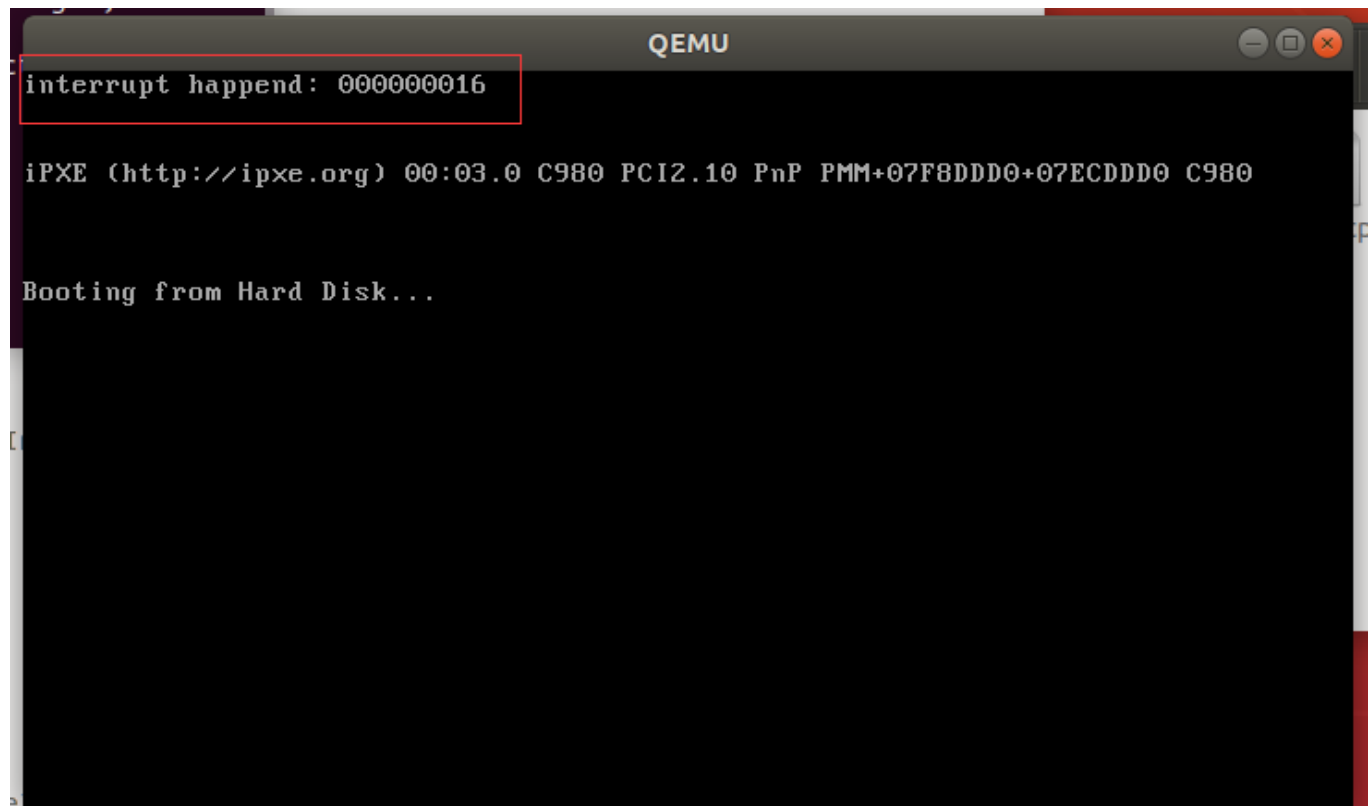
    //开中断
    asm_enable_interrupt();

    asm_halt();
}

```

各部分代码不赘述。

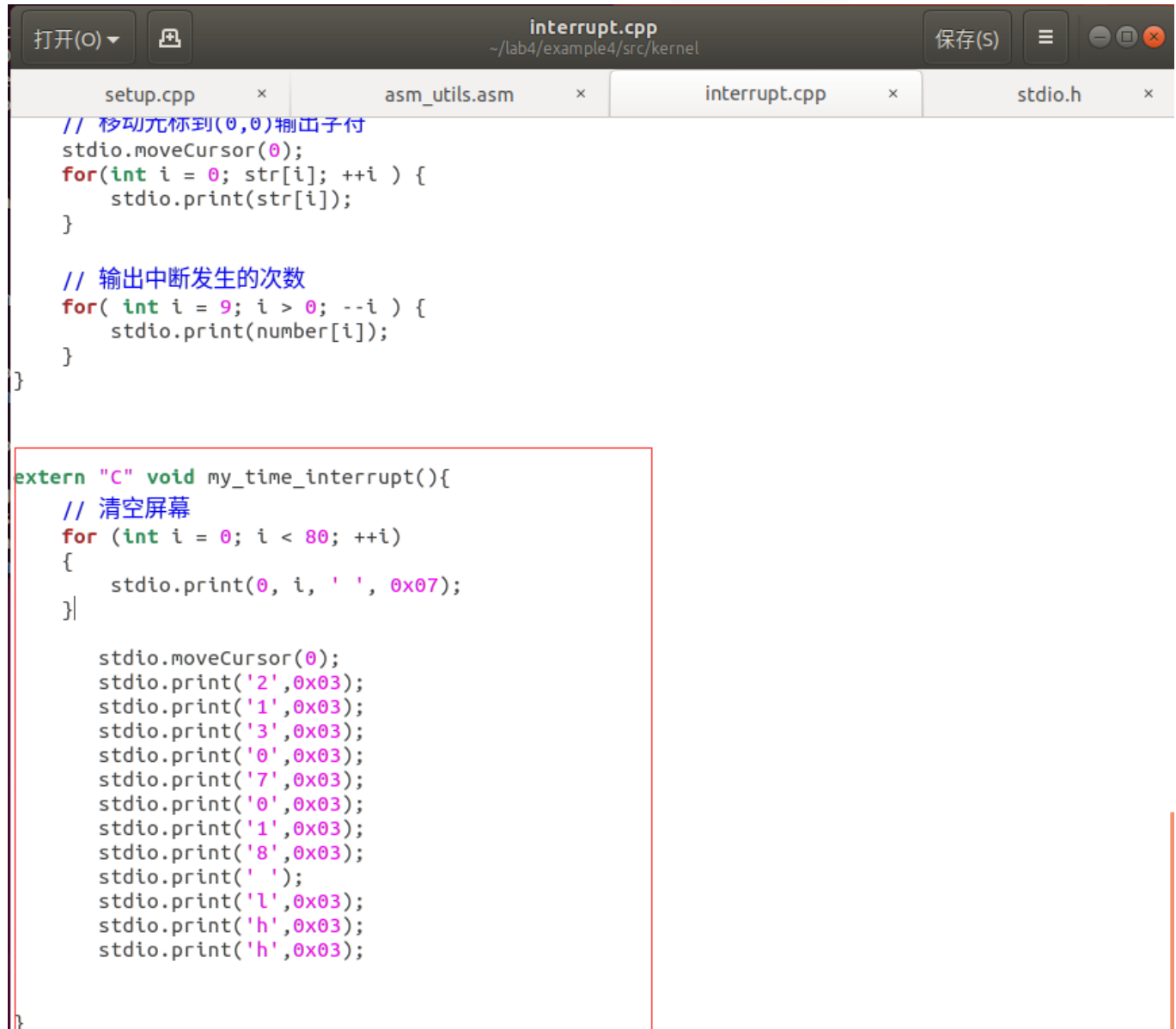
make&make run



尝试

写自己的时钟中断 (cpp)

首先写一个时钟中断处理函数 (当然也可以改原版) 在 `interrupt.cpp`



```
interrupt.cpp
~/lab4/example4/src/kernel

setup.cpp x  asm_utils.asm x  interrupt.cpp x  stdio.h x

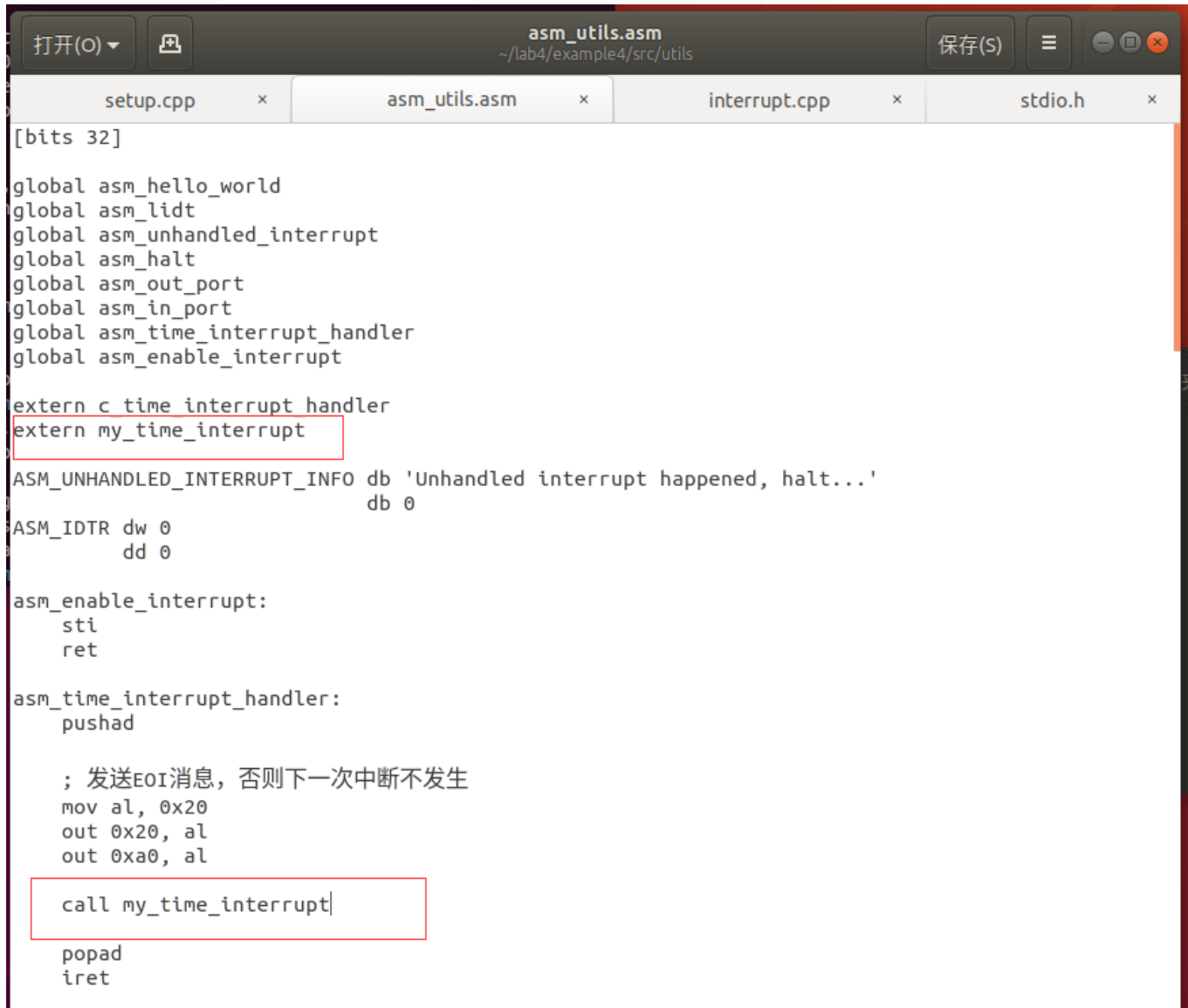
// 移动光标到(0,0)输出子付
stdio.moveCursor(0);
for(int i = 0; str[i]; ++i) {
    stdio.print(str[i]);
}

// 输出中断发生的次数
for( int i = 9; i > 0; --i ) {
    stdio.print(number[i]);
}
}

extern "C" void my_time_interrupt(){
    // 清空屏幕
    for (int i = 0; i < 80; ++i)
    {
        stdio.print(0, i, ' ', 0x07);
    }

    stdio.moveCursor(0);
    stdio.print('2', 0x03);
    stdio.print('1', 0x03);
    stdio.print('3', 0x03);
    stdio.print('0', 0x03);
    stdio.print('7', 0x03);
    stdio.print('0', 0x03);
    stdio.print('1', 0x03);
    stdio.print('8', 0x03);
    stdio.print(' ');
    stdio.print('l', 0x03);
    stdio.print('h', 0x03);
    stdio.print('h', 0x03);
}
```


再asm_utils.asm中extern它并且改掉asm_time_interrupt_handler的call内容就好。



```
[bits 32]

global asm_hello_world
global asm_lidt
global asm_unhandled_interrupt
global asm_halt
global asm_out_port
global asm_in_port
global asm_time_interrupt_handler
global asm_enable_interrupt

extern c_time_interrupt_handler
extern my_time_interrupt

ASM_UNHANDLED_INTERRUPT_INFO db 'Unhandled interrupt happened, halt...'
                               db 0

ASM_IDTR dw 0
          dd 0

asm_enable_interrupt:
    sti
    ret

asm_time_interrupt_handler:
    pushad

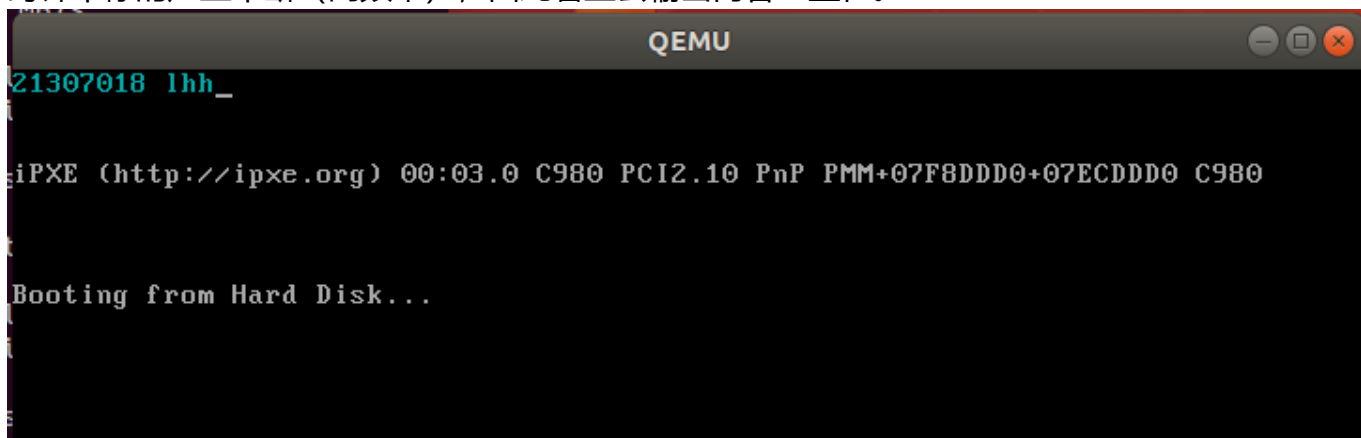
    ; 发送EOI消息, 否则下一次中断不发生
    mov al, 0x20
    out 0x20, al
    out 0xa0, al

    call my_time_interrupt|

    popad
    iret
```

make & make run效果如图

时钟不停的产生中断（高频率），因此看上去输出内容一直在。



```
QEMU

21307018 lhh_

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD+07ECDDDD C980

Booting from Hard Disk...
```