

一实验要求:

复现四个example。

二实验内容:

assignment1: 复现Example 1

- 在文件 `c_func.c` 中定义C函数 `function_from_C`。
- 在文件 `cpp_func.cpp` 中定义C++函数 `function_from_CPP`。
- 在文件 `asm_func.asm` 中定义汇编函数 `function_from_asm`，在 `function_from_asm` 中调用 `function_from_C` 和 `function_from_CPP`。
- 在文件 `main.cpp` 中调用汇编函数 `function_from_asm`。

在文件 `c_func.c` 中输入代码:

```
#include<stdio.h>
void function_from_C()
{
    printf("This is C function. \n");
    return ;
}
```

在文件 `cpp_func.cpp` 中输入代码:

```
#include<iostream>
extern "C" void function_from_CPP() {
    std::cout << "This CPP function. \n" ;
    return ;
}
```

在文件 `asm_func.asm` 中输入代码:

```
[bits 32]
global function_from_asm
extern function_from_C
extern function_from_CPP

function_from_asm:
    call function_from_C
    call function_from_CPP
    ret
```

这里利用 `function_from_asm` 函数调用一个C函数和一个C++函数。引入了这两个函数之后，在 `function_from_asm` 里直接调用这两个函数即可。

将函数 `function_from_asm` 声明为 `global` 是为了在链接阶段能找到函数的实现。在引入函数 `function_from_c` 和 `function_from_CPP` 前加 `extern`，说明函数来自外部。

在文件 `main.cpp` 中输入代码:

```
#include<iostream>
extern "C" void function_from_asm();
int main()
{
    std::cout<<"call function from assembly. " << std::endl;
    function_from_asm();
    std::cout << "Done!" << std::endl;
    return 0;
}
```

引入函数function_from_asm, 并记得在其前面加上extern, 说明此函数来自外部。引入之后在main中直接调用即可。

编译过程:

首先将这4个文件统一编译成可重定位文件即 .o 文件, 然后将这些 .o 文件链接成一个可执行文件。

输入 `gcc -m32 -c c_func.c -o c_func.o` 将文件c_function.c转换为可重定位文件, 这里经过了预处理, 编译, 汇编三个过程, 其中 `-c` 表示生成二进制文件, `-o` 生成目标文件c_func.o。

输入 `g++ -m32 -c cpp_func.cpp -o cpp_func.o`, 将文件cpp_function.cpp转换为可重定位文件。

输入 `nasm -f elf32 asm_func.asm -o asm_func.o`, 将文件asm_func.asm转换为可重定位文件, 其中 `-f elf32` 指定了nasm编译生成的文件格式是 ELF32 文件格式, ELF 文件格式也就是Linux下的 .o 文件的文件格式。

输入 `g++ -m32 -c main.cpp -o main.o`, 将文件main.cpp转换为可重定位文件。

输入 `g++ -o main.out main.o c_func.o cpp_func.o asm_func.o -m32`, 进行链接, 将多个可重定位文件链接生成可执行文件。

输入 `./main.out`, 执行文件。

运行结果:

```
azhi@azhi - Virtual Box: ~/lab4$ ./main.out
call function from assembly.
This is C function.
This CPP function.
Done!
azhi@azhi - Virtual Box: ~/lab4$
```

可以看到在main中我们调用汇编语言写的函数成功, 也说明了汇编语言写的函数调用C语言和C++写的函数成功。

assignment2: 使用C/C++来编写内核

步骤:

1、首先把lab3的bootloader.asm复制到lab4的project目录下, 并在进入保护模式的代码后加上代码

```
mov eax, KERNEL_START_SECTOR
mov ebx, KERNEL_START_ADDRESS
mov ecx, KERNEL_SECTOR_COUNT

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

```

2、把lab3的bootloader.inc文件复制到lab4的project目录下，并在后面加上加载内核需要的几个常量

```

; _____kernel_____
KERNEL_START_SECTOR equ 6
KERNEL_SECTOR_COUNT equ 200
KERNEL_START_ADDRESS equ 0x20000

```

3、创建文件entry.asm，定义内核进入点。并把该文件放在src/boot/目录下。具体定义如下：

```

extern setup_kernel
enter_kernel:
    jmp setup_kernel

```

如何让enter_kernel成为内核的进入点呢？这个将在链接阶段通过把该部分代码放在内核代码的最开始部分实现。

4、用C/C++编写函数setup_kernel.代码放在目录src/kernel/下。

setup_kernel通过调用一个汇编函数实现，这个汇编函数将输出我的学号"19335030".

```

#include "asm_utils.h"
extern "C" void setup_kernel()
{
    asm_my_number();
    while(1){
    }
}

```

5、编写汇编程序，输出我的学号。文件放在目录src/utls/asm_utils.h下。这里记得把函数asm_my_number声明为global, 以便setup_kernel调用。在文件asm_utils.h中声明所有的汇编函数，这样就不用再在调用汇编函数时前面都加extern了，只要#include "asm_utils.h"即可。代码如下：

```

[bits 32]

global asm_my_number

asm_my_number:
    push eax
    xor eax, eax

    mov ah, 0x03 ;青色
    mov al, '1'
    mov [gs:2 * 0], ax

```

```

mov al, '9'
mov [gs:2 * 1], ax

mov al, '3'
mov [gs:2 * 2], ax

mov al, '3'
mov [gs:2 * 3], ax

mov al, '5'
mov [gs:2 * 4], ax

mov al, '0'
mov [gs:2 * 5], ax

mov al, '3'
mov [gs:2 * 6], ax

mov al, '0'
mov [gs:2 * 7], ax

pop eax
ret

```

6、编译

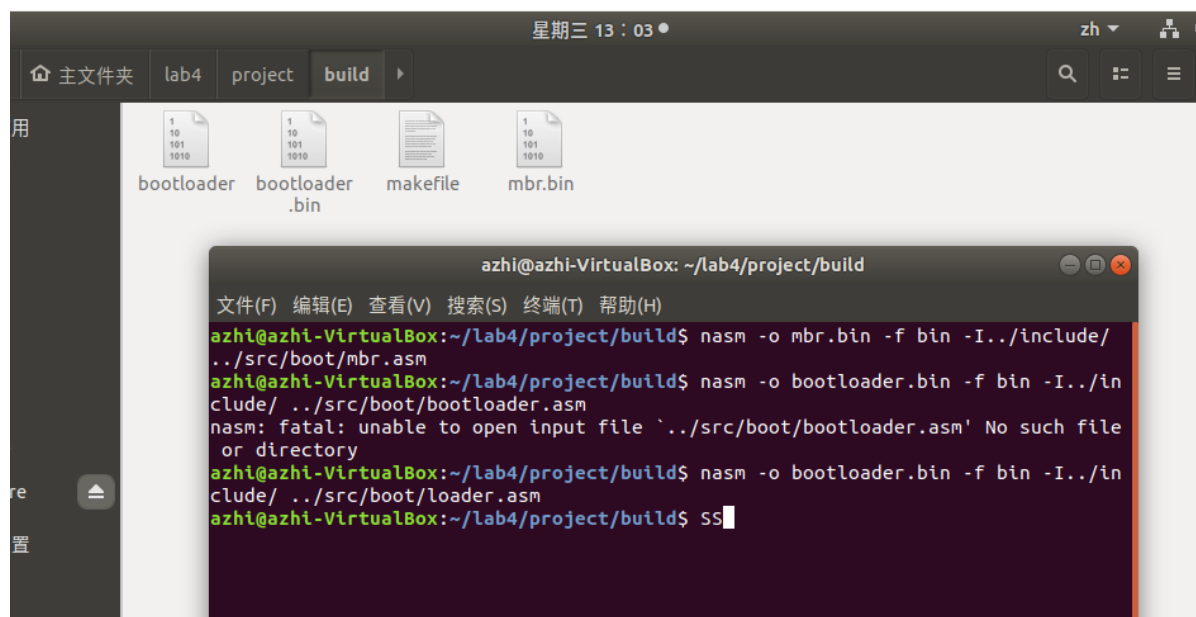
在build文件夹下开始编译，先编译MBR，

```
nasm -o mbr.bin -f bin -I../include/ ../src/boot/mbr.asm
```

编译loader.asm

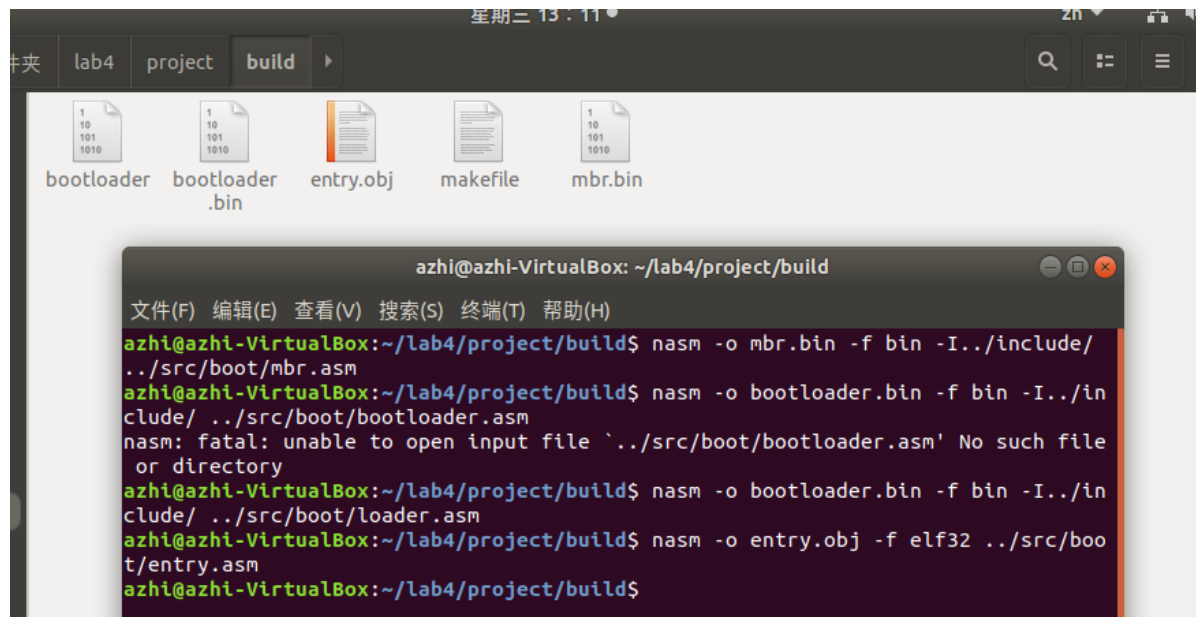
```
nasm -o bootloader.bin -f bin -I../include/ ../src/boot/loader.asm
```

如图



接着编译内核代码：

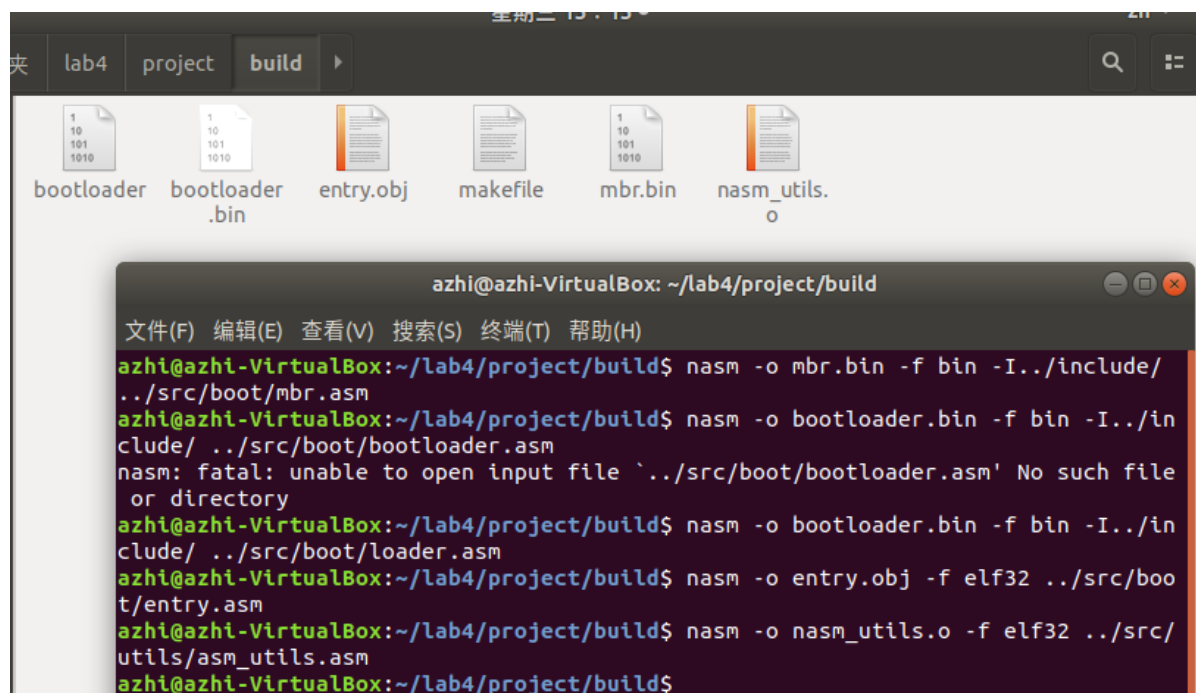
先编译 entry.asm,



The screenshot shows a file manager window with the following files: bootloader, bootloader.bin, entry.obj, makefile, and mbr.bin. Below it, a terminal window titled 'azhi@azhi-VirtualBox: ~/lab4/project/build' displays the following commands and output:

```
azhi@azhi-VirtualBox:~/lab4/project/build$ nasm -o mbr.bin -f bin -I../include/
../src/boot/mbr.asm
azhi@azhi-VirtualBox:~/lab4/project/build$ nasm -o bootloader.bin -f bin -I../in
clude/ ../src/boot/bootloader.asm
nasm: fatal: unable to open input file '../src/boot/bootloader.asm' No such file
or directory
azhi@azhi-VirtualBox:~/lab4/project/build$ nasm -o bootloader.bin -f bin -I../in
clude/ ../src/boot/loader.asm
azhi@azhi-VirtualBox:~/lab4/project/build$ nasm -o entry.obj -f elf32 ../src/boo
t/entry.asm
azhi@azhi-VirtualBox:~/lab4/project/build$
```

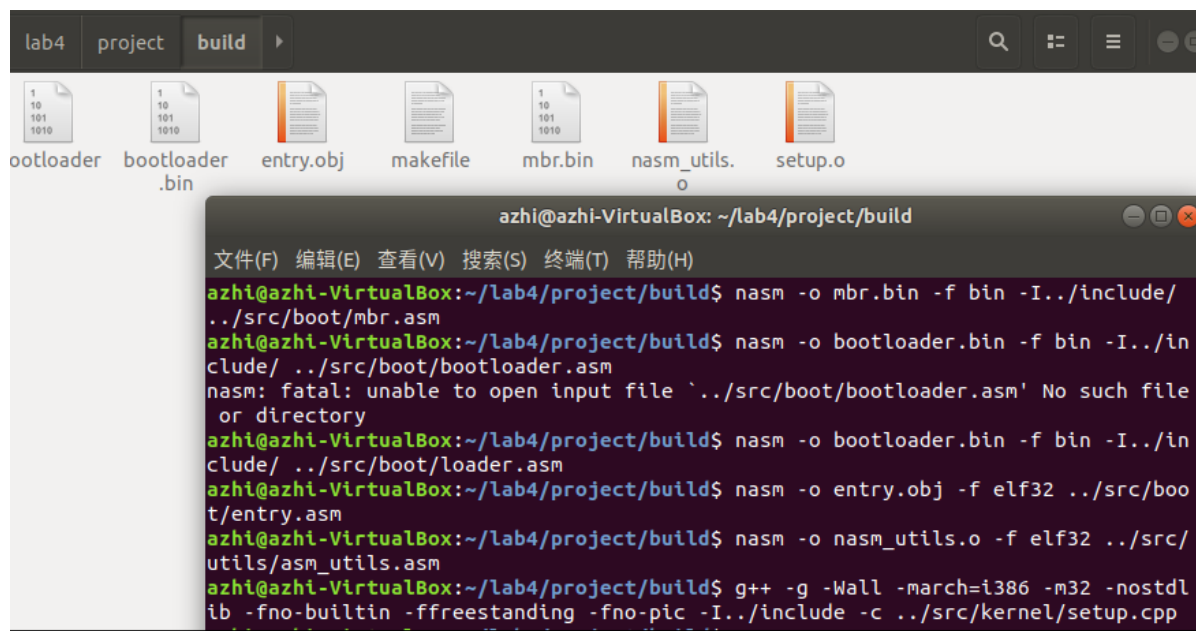
再编译 asm_utils.asm



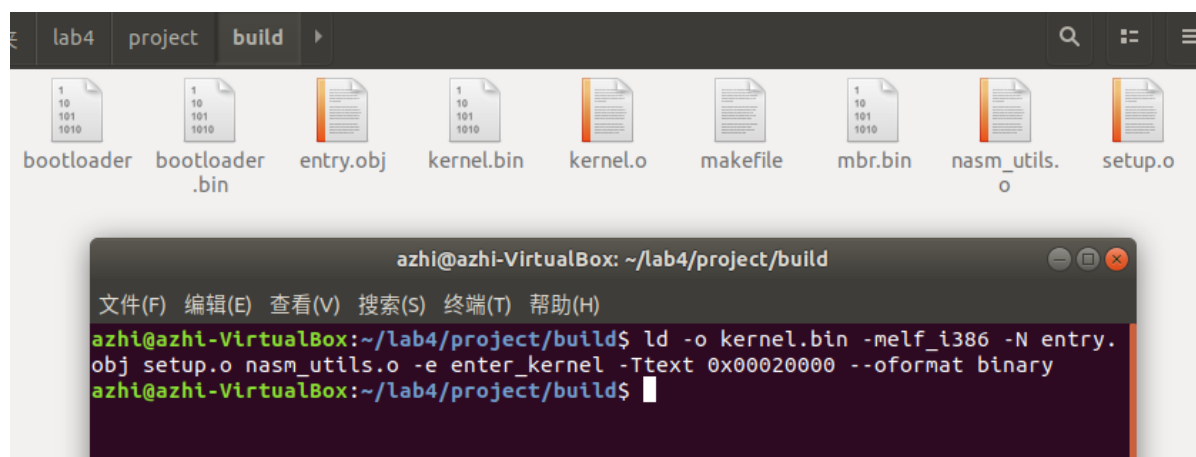
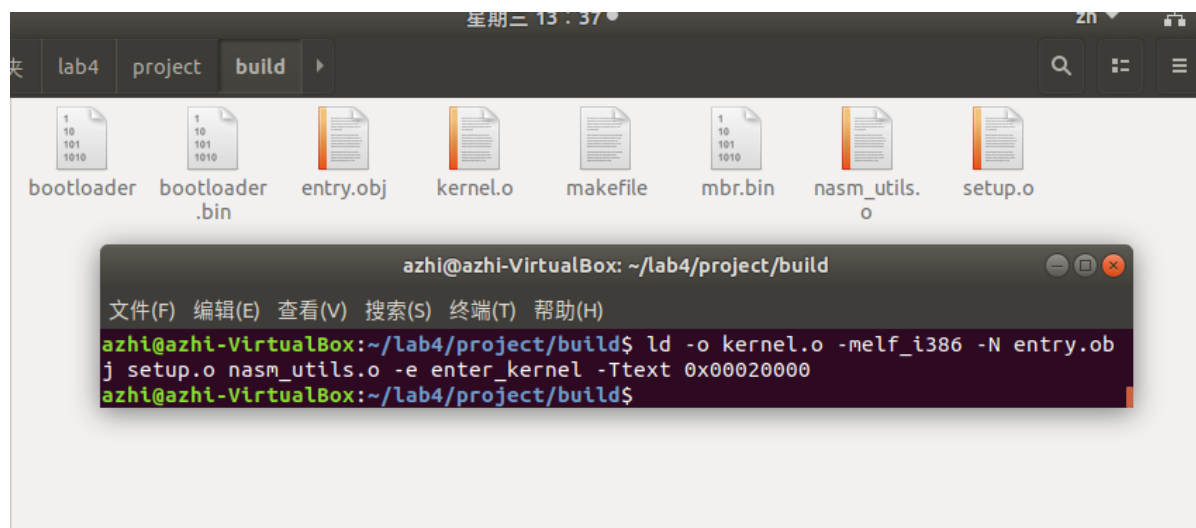
The screenshot shows a file manager window with the following files: bootloader, bootloader.bin, entry.obj, makefile, mbr.bin, and nasm_utils.o. Below it, a terminal window titled 'azhi@azhi-VirtualBox: ~/lab4/project/build' displays the following commands and output:

```
azhi@azhi-VirtualBox:~/lab4/project/build$ nasm -o mbr.bin -f bin -I../include/
../src/boot/mbr.asm
azhi@azhi-VirtualBox:~/lab4/project/build$ nasm -o bootloader.bin -f bin -I../in
clude/ ../src/boot/bootloader.asm
nasm: fatal: unable to open input file '../src/boot/bootloader.asm' No such file
or directory
azhi@azhi-VirtualBox:~/lab4/project/build$ nasm -o bootloader.bin -f bin -I../in
clude/ ../src/boot/loader.asm
azhi@azhi-VirtualBox:~/lab4/project/build$ nasm -o entry.obj -f elf32 ../src/boo
t/entry.asm
azhi@azhi-VirtualBox:~/lab4/project/build$ nasm -o nasm_utils.o -f elf32 ../src/
utils/asm_utils.asm
azhi@azhi-VirtualBox:~/lab4/project/build$
```

编译setup.cpp



链接生成的可重定位文件 `kernel.bin` 和 `kernel.o`



这里 `kernel.o` 仅用在gdb的debug过程中。

然后，用 `dd` 命令将 `mbr.bin`，`bootloader.bin` 和 `kernel.bin` 写入硬盘，并启动

结果：

```
azhi@azhi-VirtualBox: ~/lab4/project/build
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
azhi@azhi-VirtualBox:~/lab4/project/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, 0.000225592 s, 727 kB/s
azhi@azhi-VirtualBox:~/lab4/project/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
281 bytes copied, 0.000383746 s, 732 kB/s
azhi@azhi-VirtualBox:~/lab4/project/build$ dd if=mbr.bin of=../run/hd.img bs=51
2 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, 0.000252593 s, 2.0 MB/s
azhi@azhi-VirtualBox:~/lab4/project/build$ qemu
serial null -parallel stdio -no-reboot
WARNING: Image format was not specified for '...', ipxe (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD0+07
raw.
Automatically detecting the format is c
operations on block 0 will be restricted.
Specify the 'raw' format explicitly to
19335030(version 1.10.2-1ubuntu1)
QEMU
Booting from Hard Disk...
```

可以看到我的学号显示。

由于这样编译链接步骤繁杂，用了makefile就会方便很多，下面写一个makefile用于简化编译过程。

```
ASM_COMPILER = nasm
C_COMPLIER = gcc
CXX_COMPLIER = g++
CXX_COMPLIER_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/

mbr.bin : $(SRCDIR)/boot/mbr.asm
    $(ASM_COMPILER) -o mbr.bin -f bin -I$(INCLUDE_PATH)/ $(SRCDIR)/boot/mbr.asm

bootloader.bin : $(SRCDIR)/boot/bootloader.asm
    $(ASM_COMPILER) -o bootloader.bin -f bin -I$(INCLUDE_PATH)/
$(SRCDIR)/boot/bootloader.asm

entry.obj : $(SRCDIR)/boot/entry.asm
```

```

$(ASM_COMPILER) -o entry.obj -f elf32 $(SRCDIR)/boot/entry.asm

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

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

$(CXX_OBJ):
    $(CXX_COMPILER) $(CXX_COMPILER_FLAGS) -I$(INCLUDE_PATH) -c $(CXX_SOURCE)

asm_utils.o : $(SRCDIR)/utils/asm_utils.asm
    $(ASM_COMPILER) -o asm_utils.o -f elf32 $(SRCDIR)/utils/asm_utils.asm
clean:
    rm -f *.o* *.bin

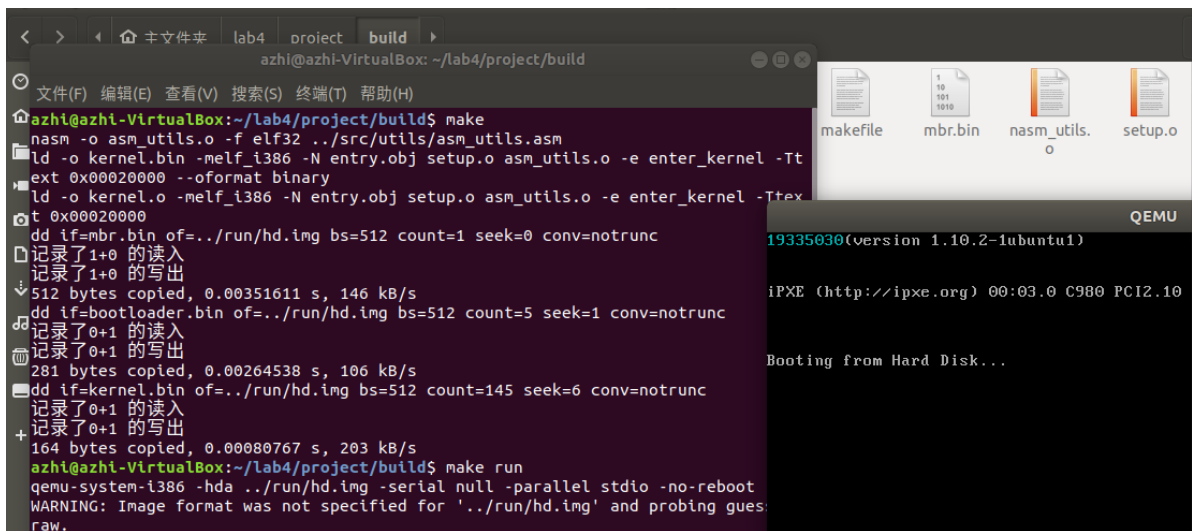
run:
    qemu-system-i386 -hda $(RUNDIR)/hd.img -serial null -parallel stdio -no-reboot

debug:
    qemu-system-i386 -S -s -parallel stdio -hda $(RUNDIR)/hd.img -serial null&
    @sleep 1
    gnome-terminal -e "gdb -q -tui -x $(RUNDIR)/gdbinit"

```

执行下列语句运行：

make 和 make run

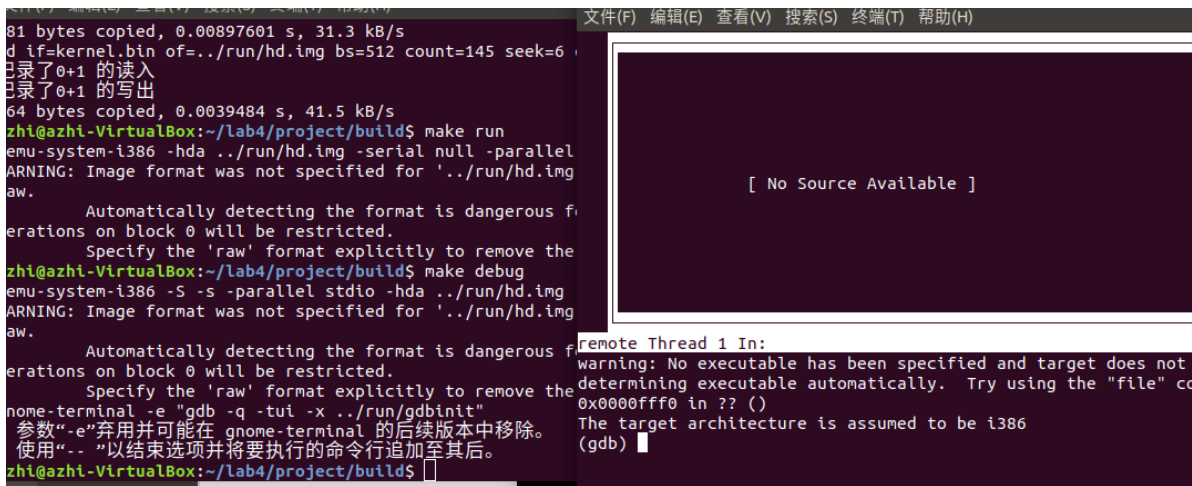


```

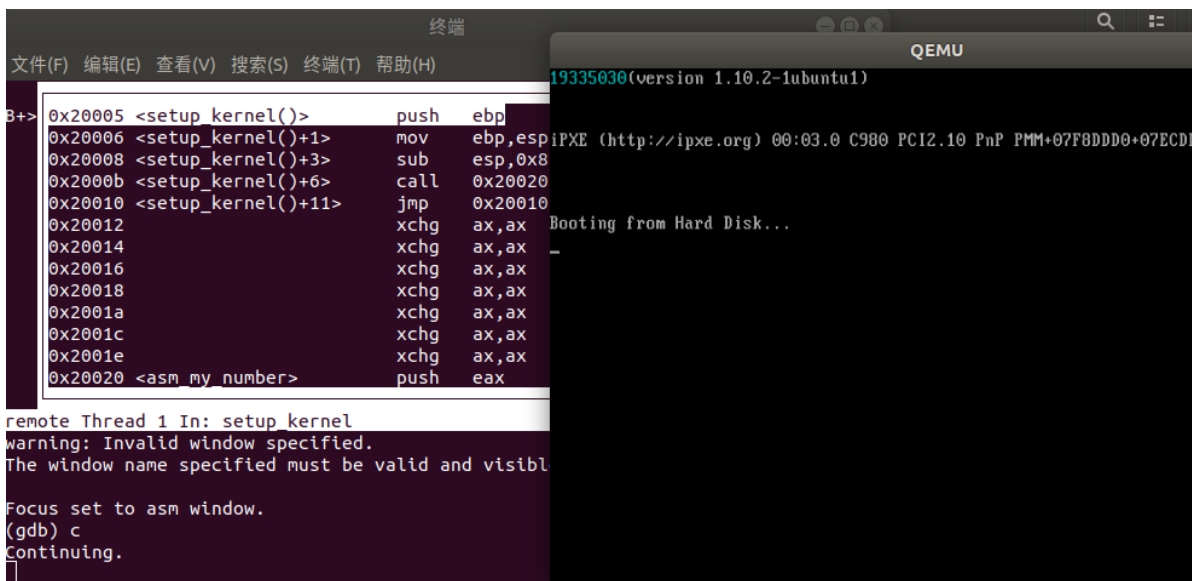
azhi@azhi-VirtualBox: ~/lab4/project/build
azhi@azhi-VirtualBox:~/lab4/project/build$ make
nasm -o asm_utils.o -f elf32 ../src/utils/asm_utils.asm
ld -o kernel.bin -melf_i386 -N entry.obj setup.o asm_utils.o -e enter_kernel -Ttext 0x00020000 --oformat binary
ld -o kernel.o -melf_i386 -N entry.obj setup.o asm_utils.o -e enter_kernel -Ttext 0x00020000
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, 0.00351611 s, 146 kB/s
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
281 bytes copied, 0.00264538 s, 106 kB/s
dd if=kernel.bin of=../run/hd.img bs=512 count=145 seek=6 conv=notrunc
记录了0+1 的读入
记录了0+1 的写出
164 bytes copied, 0.00080767 s, 203 kB/s
azhi@azhi-VirtualBox:~/lab4/project/build$ make run
qemu-system-i386 -hda ../run/hd.img -serial null -parallel stdio -no-reboot
WARNING: Image format was not specified for '../run/hd.img' and probing guess
raw.
QEMU
19335030(version 1.10.2-1ubuntu1)
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10
Booting from Hard Disk...

```

用命令 make debug 进行调试：



在弹出来的debug窗口进行调试，



assignment3: 中断的处理——初始化IDT

目的：通过实现硬中断的处理来了解保护模式下的中断处理机制。再实现用软中断调用硬中断。

我们将初始化256个中断的中断处理程序，且都是通过向栈中压入 0xdeadbeef 后做死循环实现。

初始化中断向量表主要要完成以下三件事情：

- ①确定IDT的地址；
- ②定义中断默认处理函数；
- ③初始化256个中断描述符；

步骤1:

我们将IDT的地址设为 0x8880。为了方便，先在文件 include/os_constant.h 下定义几个需要用到的常量：

```

#ifndef OS_CONSTANT_H
#define OS_CONSTANT_H

#define IDT_START_ADDRESS 0x8880
#define CODE_SELECTOR 0x20

#endif

```

这里定义了两个常量，分别是IDT的起始地址 `IDT_START_ADDRESS` 为 `0x8880` 和代码段选择子 `CODE_SELECTOR` 为 `0x20`。

步骤2:

在目录 `include/os_type.h` 下定义直观的数据类型的别名，如下

```

#ifndef OS_TYPE_H
#define OS_TYPE_H

typedef unsigned char byte;
typedef unsigned char uint8;

typedef unsigned short unit16;
typedef unsigned short word;

typedef unsigned int unit32;
typedef unsigned int unit;
typedef unsigned int dword;

#endif

```

步骤3:

为了能抽象化描述中断处理模块，在目录 `include/interrupt.h` 下定义一个 `InterruptManager` 类，且类中包含了IDT初始化函数。如下：

```

#ifndef INTERRUPT_H
#define INTERRUPT_H

#include "os_type.h"

class InterruptManager
{
private:
    // initial address of IDT
    unit32 *IDT;

public:
    InterruptManager();
    //initialize
    void initialize();
    //set index--describe_number, address--program_start_address, DPL--level
    void setInterruptDescriptor( unit32 index, unit32 address, byte DPL );
};
#endif

```

步骤4:

实现InterruptManager类中的初始化函数 `void initialize()`。

在初始化IDT之前我们要先确定IDTR的基地址和表界限。这里IDTR的32位基地址为 `0x8880`，因为每个中断描述符的大小均为8字节，故表界限为 `8*256-1=2047`。初始化IDTR需要用到汇编指令 `lidt` [tag]。指令 `lidt` 将以tag为起始地址的48字节放入到寄存器IDTR中，因此我们要用汇编语言写这个初始化函数，然后在C++中调用它。

在目录 `src/utils/asm_utils.asm` 下，实现该函数：

```
global asm_lidt
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
```

记得在 `/include/asm_utils.h` 中加入

```
extern "C" void asm_lidt(uint32 start, uint16 limit);
```

将该函数声明为外部函数。

然后我们就可以用C++实现我们的初始化函数啦！在 `/src/kernel/interrupt.cpp` 中实现，如下：

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

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

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

```

void InterruptManager::setInterruptDescriptor(uint32 index, uint32 address, byte
DPL)
{
    //low bits
    IDT[index*2]=(CODE_SELECTOR << 16) | (address & 0xffff);
    // high bits
    IDT[index*2 + 1]=(address & 0xffff0000) | (0x1 << 15) | (DPL << 13 ) | (0xe
<< 8);
}

```

其中 `setInterruptDescriptor` 函数为段描述符进行设置和定义。

段描述符由64位构成，结构如下：



即每个描述符的大小是两个 `uint32`，第 `index` 个中断描述符是 `IDT[2 * index]`，`IDT[2 * index + 1]`。

分别为段描述符的低32位和高32位赋值。

`initialize(uint32 index, uint32 address, byte DPL)` 函数先调用 `asm_lidt(uint32 start, uint16 limit)` 函数初始化IDTR，然后用 `for` 循环为每个中断描述符分配一个默认的中断处理函数。

步骤5:

定义默认的中断处理函数，放置在 `src/utls/asm_utils.asm` 目录下。

```

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 $

asm_halt:
    jmp $
```

记得声明 `global asm_unhandled_interrupt, global asm_halt` 以便其他函数调用。

`asm_interrupt_empty_handler` 首先关中断，然后输出提示字符串，最后做死循环。

步骤6:

在函数 `src/kernel/setup_kernel.cpp` 中定义初始化中断处理器。我们在这里定义一个 `InterruptManager` 的实例。如下：

```
#include "asm_utils.h"
#include "interrupt.h"

// 中断管理器
InterruptManager interruptManager;

extern "C" void setup_kernel()
{
    // 中断处理部件
    interruptManager.initialize();
    // 尝试触发除0错误
    //int a = 1 / 0;
    // 死循环
    asm_halt();
}
```

当去掉 `int a=1/0` 那一行的注释时，我们可以利用触发除0异常来验证 `asm_unhandled_interrupt` 是否正常工作。

步骤7:

在 `include/os_modules.h` 中声明这个实例，以便在其他 `cpp` 文件中使用。如下

```
#ifndef OS_MODULES_H
#define OS_MODULES_H

#include "interrupt.h"

extern InterruptManager interruptManager;

#endif
```

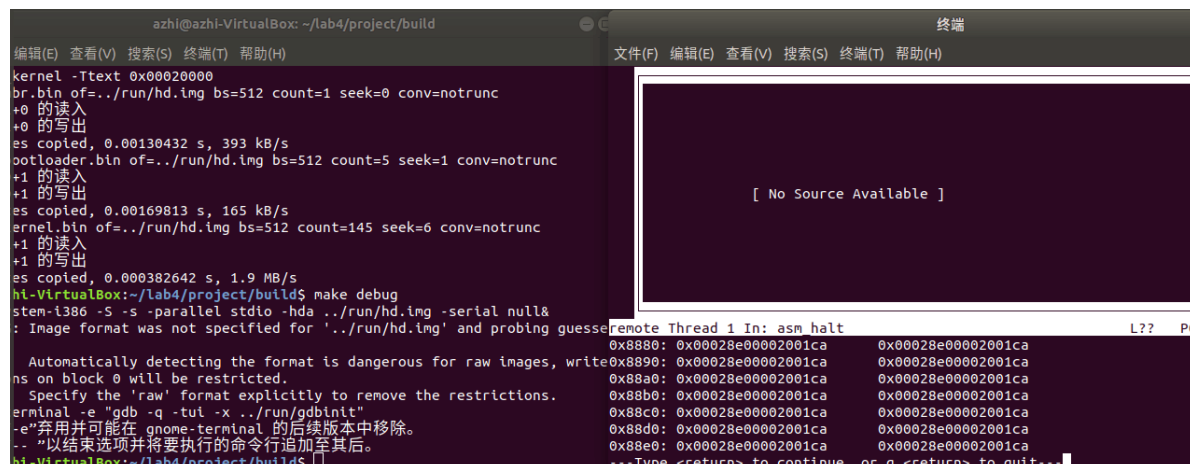
步骤8:

用 `Makefile` 编译运行。在编译前一定要把 `Example2` 中生成的中间文件删除，再重新 `make`，这样生成的文件才是新的。

首先输入 `make`，进行编译。

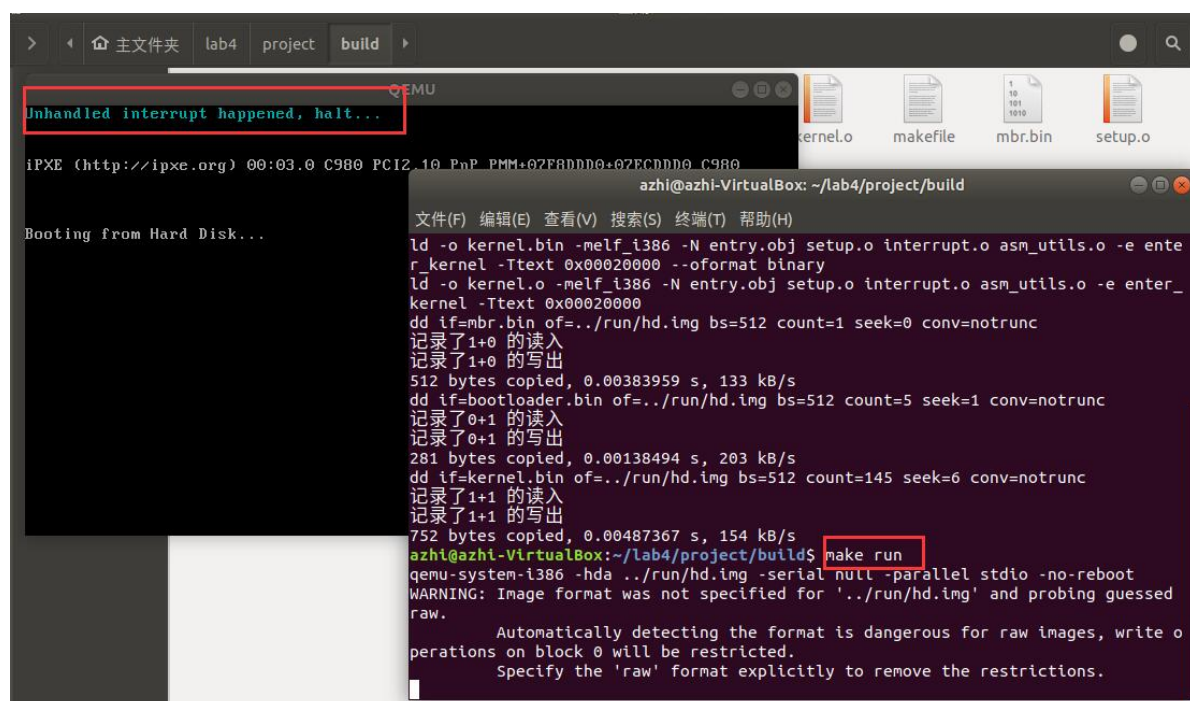
然后，输入 `make debug` 进入 `gdb` 窗口调试。

先在 gdb 窗口下让程序运行，然后，再按 `ctrl+c` 后输入 `x/256x 0x8880` 查看默认的中断描述符是否已被放入。



```
azhi@azhi-VirtualBox: ~/lab4/project/build
编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
kernel -Ttext 0x00020000
br.bin of=../run/hd.img bs=512 count=1 seek=0 conv=notrunc
+0 的读入
+0 的写出
es copied, 0.00130432 s, 393 kB/s
bootloader.bin of=../run/hd.img bs=512 count=5 seek=1 conv=notrunc
+1 的读入
+1 的写出
es copied, 0.00169813 s, 165 kB/s
ernel.bin of=../run/hd.img bs=512 count=145 seek=6 conv=notrunc
+1 的读入
+1 的写出
es copied, 0.000382642 s, 1.9 MB/s
hi-VirtualBox:~/lab4/project/build$ make debug
stem-i386 -S -s -parallel stdio -hda ../run/hd.img -serial null&
: Image format was not specified for '../run/hd.img' and probing guessed
Automatically detecting the format is dangerous for raw images, write o
ns on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
terminal -e "gdb -q -tui -x ../run/gdbinit"
-e"弃用并可能在 gnome-terminal 的后续版本中移除。
-- "以结束选项并将要执行的命令行追加至其后。
hi-VirtualBox:~/lab4/project/build$
remote Thread 1 In: asm halt
L?? PC
0x8880: 0x00028e00002001ca 0x00028e00002001ca
0x8890: 0x00028e00002001ca 0x00028e00002001ca
0x88a0: 0x00028e00002001ca 0x00028e00002001ca
0x88b0: 0x00028e00002001ca 0x00028e00002001ca
0x88c0: 0x00028e00002001ca 0x00028e00002001ca
0x88d0: 0x00028e00002001ca 0x00028e00002001ca
0x88e0: 0x00028e00002001ca 0x00028e00002001ca
---Type <return> to continue, or q <return> to quit---
```

然后，我们在 `src/kernel/setup.cpp` 下加上除0语句，再一次 `make` 和 `make run` 查看是否出现中断。记得 `make` 之前一定要把之前生成的中间文件删除，否则运行的仍然是没有除0语句的操作，这样将无法看到中断。



```
> < 主文件夹 lab4 project build
Unhandled interrupt happened, halt...
IPXE (http://ipxe.org) 00:03:0 C980 PCI2.10 PnP PMM+07F8DD00+07ECDD00 C980
Booting from Hard Disk...
azhi@azhi-VirtualBox: ~/lab4/project/build
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
ld -o kernel.bin -melf_i386 -N entry.obj setup.o interrupt.o asm_utils.o -e ente
r_kernel -Ttext 0x00020000 --oformat binary
ld -o kernel.o -melf_i386 -N entry.obj setup.o interrupt.o asm_utils.o -e ente
kernel -Ttext 0x00020000
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, 0.00383959 s, 133 kB/s
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
281 bytes copied, 0.00138494 s, 203 kB/s
dd if=kernel.bin of=../run/hd.img bs=512 count=145 seek=6 conv=notrunc
记录了1+1 的读入
记录了1+1 的写出
752 bytes copied, 0.00487367 s, 154 kB/s
azhi@azhi-VirtualBox:~/lab4/project/build$ make run
qemu-system-i386 -hda ../run/hd.img -serial null -parallel stdio -no-reboot
WARNING: Image format was not specified for '../run/hd.img' and probing guessed
raw.
Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
```

可以看到，出现了中断提示语句，因此中断被正常调用了。至此，我们已经完成了IDT的初始化。

assignment4: 时钟中断

对8259A进行编程，添加处理实时钟中断的函数，函数在第一行显示目前中断发生的次数。

主要步骤有：

1、在中断控制器 `InterruptManager` 中加入时钟中断的成员变量和函数。

加入成员变量 `uint32 IRQ0_8259A_MASTER` 表示主片中断起始向量号，`uint32 IRQ0_8259A_SLAVE` 表示从片中断起始向量号。

加入函数 `void initialize8259A()` 用于初始化芯片8259A。在该函数中依次通过向8259A的特定端口发送四个初始化命令字ICW1~ICW4。四个初始化命令字必须严格按照顺序发送。

发送初始化命令字通过调用汇编函数下对 `out` 指令进行封装实现。

在目录 `/src/utils/asm_utils.asm` 中编写函数 `asm_out_port` 对指令 `out` 进行封装。记得声明 `global asm_out_port`。

然后在函数 `initialize8259A` 中调用 `asm_out_port` 进行ICW1~ICW4的初始化。

加入函数 `void setInterruptDescriptor()` 开启时钟中断。

加入函数 `void enableTimeInterrupt()` 禁止时钟中断。

加入函数 `void setTimeInterrupt(void *handler)` 用于设置时钟中断处理函数。

2、实现中断处理函数。

首先，因为我们需要在屏幕上进行输出，于是在这里先实现一个能够处理屏幕输出的类。

在目录 `/src/include/stdio.h` 做相关函数的声明。

这里主要实现了对字符串和光标的处理。主要实现的功能有①打印特定颜色的字符②打印特定颜色的光标③移动光标位置④获取光标位置⑤滚屏。

屏幕像素为 25×80 ，因此字符或光标的位置x不超过25，y不超过80。

各函数的实现在目录 `/src/kernel/stdio.cpp` 下。其中有用到函数 `asm_in_port` 是对汇编指令 `in` 的封装，放在目录 `/src/utils/asm_utils.asm` 下。

与光标读写相关的端口为 `0x3d4` 和 `0x3d5`，在对光标读写之前，要向端口 `0x3d4` 写入数据，表明我们操作的是光标的低8位还是高8位。写入 `0x0e`，表示操作的是高8位，写入 `0x0f` 表示操作的是低8位。如果需要读取光标，那么我们从 `0x3d5` 从读取数据；如果我们需要更改光标的位置，那么我们将光标的位置写入 `0x3d5`。

`stdio.cpp` 的具体实现见文件。

然后，在 `/src/kernel/interrupt.cpp` 中定义中断处理函数 `c_time_interrupt_handler`。

`c_time_interrupt_handler` 首先清空第一行的字符，然后对计数变量 `times` 递增1，并将其转换成字符串。

因为中断最后要用 `iret` 返回，所以只能在汇编函数里通过调用 `c_time_interrupt_handler` 来实现中断。如下：

```
asm_time_interrupt_handler:
    pushad

    nop ; 否则断点打不上去
    ; 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 c_time_interrupt_handler

    popad
    iret
```

其中 `pushad` 指令将 `EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI` 依次入栈，`popad` 进行出栈操作。

编写好了中断处理函数后，设置时钟中断的中断描述符，如下所示：

```

void InterruptManager::setTimeInterrupt(void *handler)
{
    setInterruptDescriptor(IRQ0_8259A_MASTER, (uint32_t)handler, 0);
}

```

3、对开启和关闭时钟中断的函数进行封装。

```

void InterruptManager::enableTimeInterrupt()
{
    uint8_t 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_t value;
    asm_in_port(0x21, &value);
    // 关闭时钟中断, 置1关闭
    value = value | 0x01;
    asm_out_port(0x21, value);
}

```

4、最后记得在 `setup_kernel` 中实例化 `stdio`, 并初始化内核组件, 然后开启时钟中断。

```

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();
}

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();
}

```

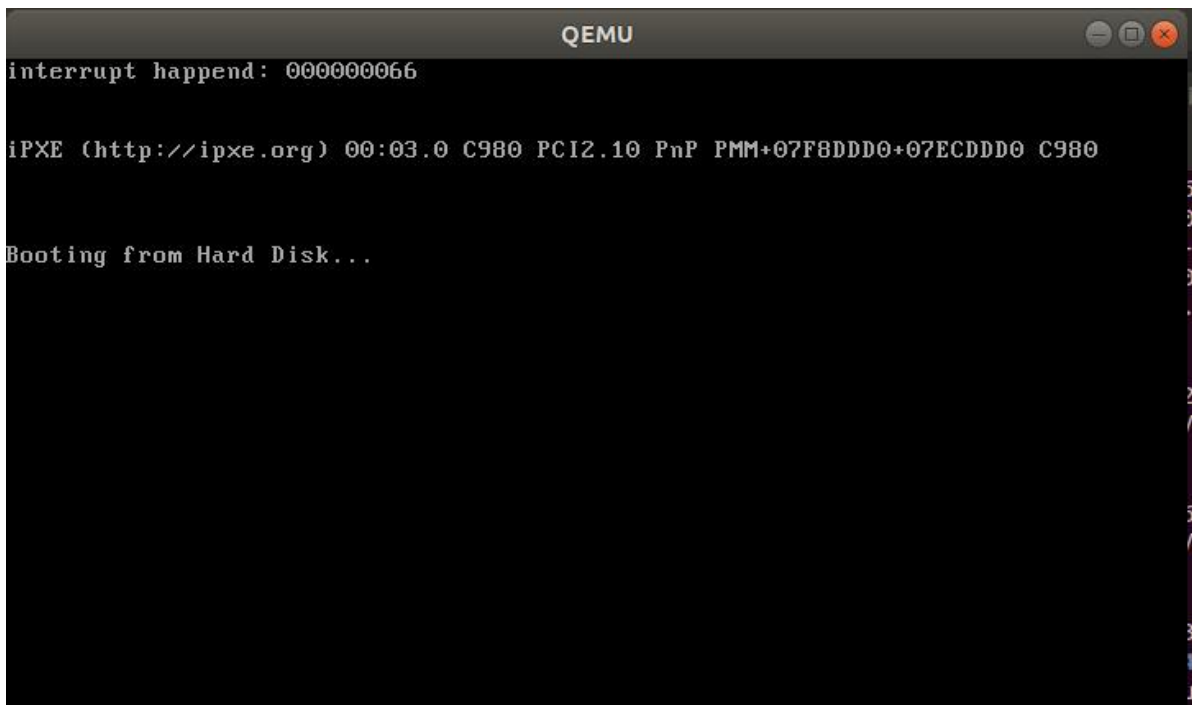
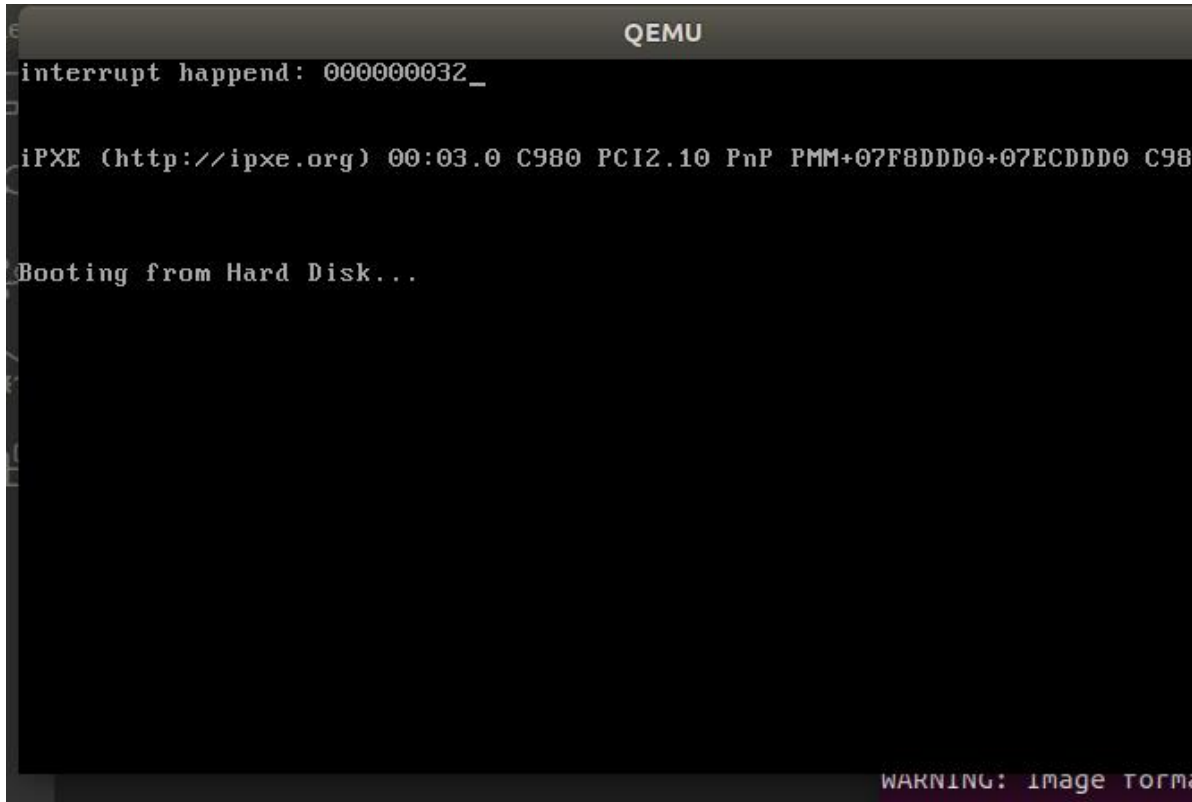
在 `include/os_modules.h` 中声明这个实例:


```
#ifndef OS_MODULES_H
#define OS_MODULES_H
#include "interrupt.h"
extern InterruptManager interruptManager;
extern STDIO stdio;
#endif
```

由于开中断的指令要用到 `sti`，因此把这条指令加在函数 `asm_enable_interrupt` 中。

5、编译运行：

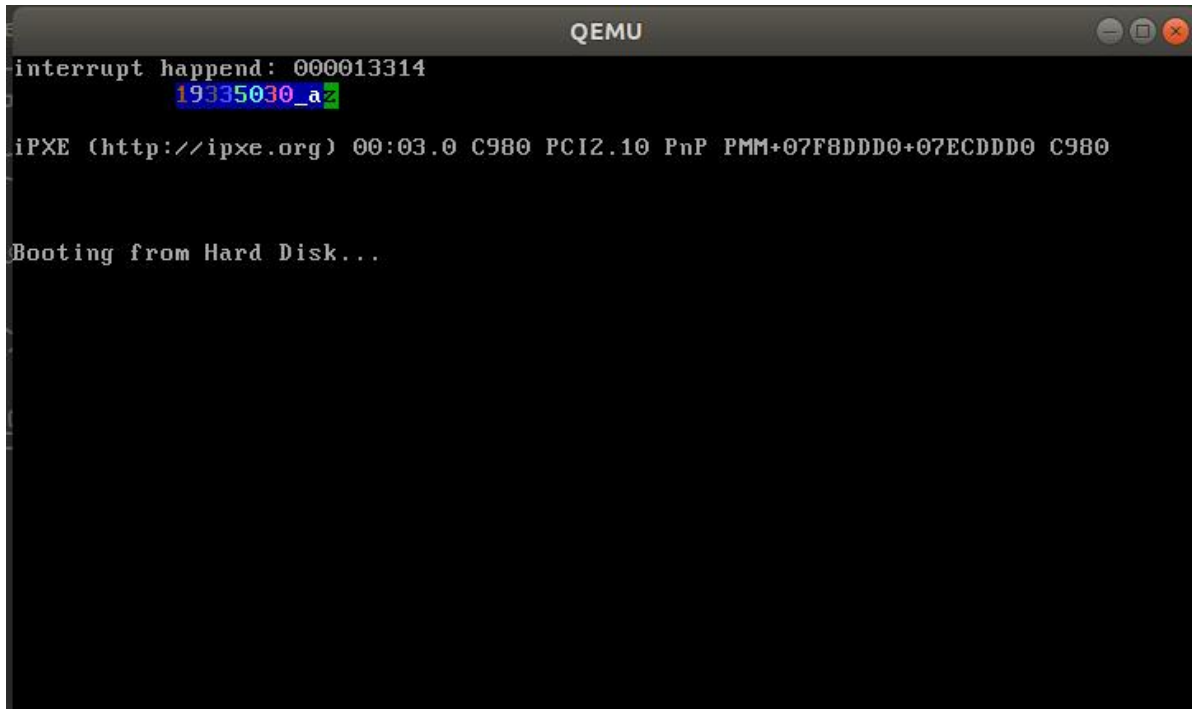
在目录 `/build/` 下打开终端，输入命令 `make` 和 `make run` 查看结果：



可以看到，第一行输出了中断发生的次数。

6、跑马灯显示学号姓名。

在 `interrupt.cpp` 中的 `c_time_interrupt_handler()` 中加入显示自己学号姓名的代码。然后再编译运行，查看结果。



```
QEMU
interrupt happend: 000013314
19335030_azhi
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD0+07ECDDDD0 C980

Booting from Hard Disk...
```

如图示，在第二行中显示了我的学号和姓名，并且它会向前滚动。

代码如下：

```
char num_name[] = "19335030_azhi: ";
int flag = times%80;
while(1){

    for( int j =0; j < 330; j++ ){
        for (int k = 0; k < 80; ++k)
        {
            stdio.print(1, k, ' ', 0x07);
        }
        int k = j/5; //控制输出速度
        stdio.moveCursor(1, k);
        for(int i = 0; i < 15; i++)
        {
            stdio.print(num_name[i], 0x00+i+flag);
        }

    }
}
```

三实验感想：

此次实验内容很多，要花费很多时间才能学透这一次实验的内容。另外一个难题是好多汇编代码看不是很懂，理解起来会比较慢。这也增加了实验难度。但是实验教程写得很详细，再参考一下其他资料，就可以完成实验了。